Errata

Corrections to the printed IntraBuilder documentation are detailed in the file ERRATA.HLP, available for download from the IntraBuilder home site at http://www.borland.com/intrabuilder.

You can also download the latest build of the IntraBuilder online Help system from the product home site.

Language Reference

Views and Tools

Quick start notes
Table Designer
Form Designer
Script Pad
The Inspector
Method Editor
Report Designer introduction
Component Palette
Field Palette
The Home Page Form
Overview of IntraBuilder security

Topics found

close() [File]

close() [Form]

Topics found

copy() [UpdateSet]

copy() [File]

Topics found

count() [Array]

count() [Rowset]

Topics found

class Database

database [Query,StoredProc]

Topics found

Topics found

delete() [Rowset]
delete() [UpdateSet]
delete() [Array]
delete() [File]

Topics found

class Form

form [all form components]

Topics found

handle [File]

handle [Database, Query, Rowset, Session, StoredProc]

Topics found

indexName [Rowset]

indexName [UpdateSet]

Topics found

insert() [Array]

INSERT [Local SQL]

Topics found

left() [StringEx]

left [all form objects]

Topics found

length [Array]
length [Field]
length [String, StringEx]

Topics found

onChange [ListBox, Select, Text, TextArea]

onChange [Field]

Topics found

open() [File]

open() [Form]

Topics found

params [ActiveX, JavaApplet]

params [Query, StoredProc]

Topics found

Topics found

right() [StringEx]

right [Rule]

Topics found

class Rowset

rowset [Query, StoredProc]

Topics found

class Select

SELECT [Local SQL]

Topics found

class Session

session [Database, Query, StoredProc]

Topics found

size() [File]

size [Rule]

Topics found
class StreamSource
streamSource [StreamFrame]

Topics found

substring() [String, StringEx]

Substring [Local SQL]

Topics found

class Text

text [CheckBox, Radio, Text]

Topics found

type [Field]

type [Parameter]

Topics found

update() [UpdateSet]

UPDATE [Local SQL]

Topics found

value [CalcField]
value [Field]
value [Parameter]

Introducing IntraBuilder

Welcome to Borland IntraBuilder,[a] the easiest and most efficient way to create and maintain live data solutions on a World Wide Web-based network.

This booklet introduces you to the program, its parts, and its basic operation. It also introduces you to the fundamental concepts behind the Internet and intranets, and describes how IntraBuilder works with these technologies to enable you to produce, deliver, and maintain custom interactive data applications on any TCP/IP network (Transmission Control Protocol/Internet Protocol, the type of network connection used on the Internet).

You'll see how fast and easy it is to create a basic IntraBuilder application, run it on your server, and view it through any Web browser. Finally, we'll show you how to get the most out of the comprehensive IntraBuilder online Help system.

First, let's deal with a couple of big questions and then look at some of the things you can do with IntraBuilder.

Introducing IntraBuilder, cont'd: **Why IntraBuilder? Why now?**

Why IntraBuilder? Why now?

Most business people and others who work anywhere near a computer have a pretty good idea what the Internet is. Universal Resource Locators (URLs) like “http:// www.borland.com” appear regularly in advertising, sometimes as the only contact information. Corporate URLs and E-mail addresses are business card essentials.

The Internet has become so commonplace, it seems, that many people are already taking it for granted as a sort of universal information provider, a limitless library of data that one can access any time using any Web browser on any machine that has a modem or a company network connection to the outside world.

The trouble with the “universal information provider” notion is that the Internet really hasn’t been very good at providing information. Much of the data that flows through the thousands of worldwide network hubs that make up the Internet is static. And while many of these static pages provide useful information, the assemble-and-publish model upon which they’re based is of little use to companies and organizations that have vast database stores and a need to get that information out to eager users inside and outside of the company.

Mere publishing can’t handle the volume. Search engines, as fast and efficient as they’ve become, are themselves static in design and require cumbersome low-level programming to create and modify.

Intranets—the mini-Internets that now exist within companies around the world—have been hardest hit by the lack of dynamic data delivery systems for the Web.
At many of these companies the hardware is in place, robust TCP/IP network connections are established, and a new group of system administration professionals, known as “webmasters,” continue to search for ways to harness this vibrant technology, to find a truly efficient, reliable, and low-maintenance way to make existing legacy data and constantly changing updates available to anyone with a browser.

The most common solution has generally been to try to develop custom applications using traditional programming tools. Some companies and Web application developers have achieved the goal to one extent or another, but state management, application partitioning, security, and other major stumbling blocks have always made it a difficult chore. The cost of such development has always been high.

Introducing IntraBuilder, cont’d: **Enter IntraBuilder**

Enter IntraBuilder

The IntraBuilder solution meets the needs of intranet developers, webmasters, and database administrators. It can serve the needs of a wider Internet audience as well, but, as a complete database application programming environment, it is most at home where the data lives—within the firewalls of an intranet.

What used to take experienced intranet developers days to create now takes minutes to develop and deliver with IntraBuilder. Real-time reports, updates, queries, filters, data source connections, administrative tasks, security, password protection—it's now all wrapped up in a point-and-click interface and powered by a sleek, efficient JavaScript programming engine.

With IntraBuilder, the promise of the Internet and intranet as full-time, fully-enabled vehicles of live data delivery has finally arrived.

IntraBuilder feature list

Some ideas for IntraBuilder applications

IntraBuilder is capable of doing many things for many people, but here are a few suggestions.

- Create an interactive personnel database that allows job applicants to enter and update resume information remotely.
- Create pages that allow field sales personnel to retrieve current reports on sales data, price shifts, and inventory.
- Create forms to allow field sales personnel to enter expense reports and travel data for automatic processing and storage.
- Create a meeting room organizer and booking site that displays schedules and current status information for each room.
- Create an illustrated corporate phone directory that allows employees, through record-level password protection, to change their own data.
- Create a searchable knowledge base to store employee handbook contents and other procedural materials.
- Create a threaded message application to enable interdepartmental conversation on specific issues.

IntraBuilder feature list

Here are some of the features that make IntraBuilder the easiest, most efficient, and most powerful Web-based network application development package you can use.

If you're new to Web-based application development

For advanced users, administrators, and developers

If you're new to Web-based application development

- Easy to use. IntraBuilder's easy to use point-and-click, drag-and-drop integrated design environment lets you create applications in minutes instead of days.
- Little knowledge required. No knowledge of HTML, Web protocols, network programming, or programming languages is required. In fact, you can use IntraBuilder to create applications without ever having to program a line of code. If you do like to get "under the hood," however, you're in for a treat as well: Developers can delve into the richness and power of the underlying JavaScript code any time using IntraBuilder's full-featured program editor. Then there's the visual two-way tools (described later) that let you mix your programming styles any way you like.
- Experts let you create and deliver form-based applications, reports, tables, and home pages with guided, step-by-step control over design and content. You can also customize the experts to create templates for consistent table structuring and design throughout your site. The experts are
- Form Expert, for creating data-entry forms. Choose from a complete set of data navigation buttons, graphics, colors, fonts, and other design and layout elements.
- Home Page Expert lets you create and manage Web pages. Add logos, text, graphics, navigation buttons, and internal or external links by choosing options as you move through the expert's steps. Your finished pages can be placed anywhere in your site structure as stand-alone elements or used as launchpads for other forms and applications.
- **Report Expert** guides you through the design and delivery of every detail, grouping, summary, and totals sections of a report. This expert provides a quick and easy way to deliver live complex statistical information.
- **Table Expert** provides field and table templates. You can mix and match fields, and build tables from most popular database formats including dBASE,¨ Paradox,¨ InterBase,ª Access, MS SQL Server, Sybase, and Oracle using a choice of built-in drivers or via ODBC or SQL-Link drivers.
- **Visual Property Builders**. These special dialog boxes let you fine-tune options for specific form and element attributes. IntraBuilder has more than 20 builders that let you set and adjust attributes in forms, reports, home pages, and other components.
- **Designers** give you even more control and flexibility over your design and development tasks. An alternative to the experts when creating new forms, tables, reports, and other components, the designers allow you to drag and drop controls and fields from palettes and use the built-in object Inspector and the Visual Property Builders to specify properties, events, and methods. The designers also come into play when modifying existing components. As with the experts, there are designers for forms, tables, reports, and home pages. Of course, you can also use the experts and designers together, first using the experts to create new components, then switching to the designers to customize them.
- File drag and drop. IntraBuilder knows its environment. Have a table at hand in the IntraBuilder Explorer, or your Windows Explorer or Find box (or File Manager in NT 3.51)? Just drag the file right onto a form, then use the Form Designer to specify its properties.
- Java applets and ActiveX controls. The Form Designer provides visual tools for integrating Java applets and ActiveX controls, extending your application's capabilities to include the latest gizmos and gadgets as soon as they're offered.
- Templates and custom styles. The Scheme dialog box lets you choose from dozens of supplied design templates or create new templates of your own. Schemes let you control colors, fonts, styles, background bitmaps, and other aspects of your designs, allowing you to produce a consistent look and feel to forms and pages throughout your site.
- Most image formats supported. IntraBuilder supports most popular Web and Windows image formats. The IntraBuilder Designer accepts any of the types listed later in this section, automatically converting files when needed to formats that are acceptable to the requesting Web browser.

For advanced users, administrators, and developers

- Full-featured **editors.** The Script Pad, Text Editor, Program Editor, and Method Editor are full-featured, fully customizable word processors designed to enhance your programming productivity. (Of course, you can use the Text Editor for any non-IntraBuilder job as well.) In Windows 95 and Windows NT 4.0, you can even drag code snippets onto the desktop, then drag them back into IntraBuilder when needed.
- Automatic **state management** allows standard database operations to coexist with the “stateless” demands of Web browser access.
- Automatic JavaScript. Components of an IntraBuilder application are automatically saved in script files using an extended version of the industry-standard JavaScript language. The JavaScript code can be edited directly using IntraBuilder’s powerful script-editing capabilities. Using the designers and Visual Property Builders as two-way tools, you can mix direct code entry with point-and-click design elements.
- **ANSI language drivers** are available to ease the process of international enabling.
- Advanced client/server features. IntraBuilder brings advanced client-server features to the rapid application development (RAD) arena, providing all the tools you need to develop Web applications that tie into SQL servers, ODBC data sources, and native Borland Database Engine (BDE) tables.
- High-performance SQL-Link drivers are supplied for IBM DB2, Oracle, Sybase, Informix, InterBase, and the Microsoft SQL Server. The Borland Database Engine also has an ODBC socket that works with popular desktop formats such as Access and Approach. Native drivers for DBF and DB tables are also provided.
- Concurrency control. IntraBuilder offers a wide array of concurrency control when connected to database servers such as Oracle, Sybase, and Microsoft SQL Server. While the program provides an automatic optimistic locking scheme, developers can easily configure pessimistic locking through a variety of methods using IntraBuilder’s powerful database object model.
- **Administration tools** let database administrators create tables on remote servers.
- **Data encryption tools** are available for DBF and DB tables, allowing administrators to set both table and field level privileges.
- **Visual referential integrity** tools are available for any servers attached via SQL Links and most ODBC data sources.
- **Field modification.** The Table Designer contains a field inspector for defining column constraints.
- Sophisticated Report Designer. The Report Designer offers such sophisticated features as self-evaluating code blocks (which can contain any valid expression, from simple field references to multiple method calls), design-time HTML tag evaluation and stream frames. The Report Designer also lets you control alignment, margins, leading, tracking, colors, and borders for any HTML component in your report.
- Visual inheritance and subclassing. The Form Designer offers visual inheritance and visual subclassing, allowing you to easily create and save custom form classes as templates for new forms. IntraBuilder even provides a special designer, called the Custom Form Class Designer, to let you visually edit your custom form classes and provide another means of standardizing form and page design throughout your site.
- Custom components. You can derive custom components from standard controls, data access objects, or other custom components, making it easy to reuse objects. IntraBuilder automatically adds your custom components to its components palette to keep them in easy reach.
- Data access classes. IntraBuilder data access classes merge the SQL and object-oriented paradigms. You can use queries within databases within sessions. Sessions provide independent connections to tables. Each database can then connect to a different data source. The queries connect to one or more tables and provide table navigation capability.

Products and programs in your IntraBuilder package

- The IntraBuilder Designer (INTRA.EXE).
- The IntraBuilder Server, which is made up of the IntraBuilder Broker (INTRASRV.DLL and INTRASRV.ISV) and one or more IntraBuilder Agents (instances of INTRA.EXE). The IntraBuilder Server is surfaced through the Agents, which appear as minimized icons in your taskbar or desktop.
- A selection of pre-built business solutions. These applications are ready-to-run. Or, you can re-use any parts of them to help create your own applications. And you can examine and learn from the JavaScript code attached to components on the forms.
- A selection of sample tables, forms, and graphics, plus special designer forms and tables you can use for various business and personal tasks.
- The 32-bit Borland Database Engine (BDE) and configuration utility (BDECFG32.EXE).
- The Borland Personal Web Server.
- Support for Paradox, dBASE, Microsoft Access, and Microsoft FoxPro databases.
- Additional database drivers and support (IntraBuilder Professional and Client/Server editions only).
- Integrated Help system, including a full *Language Reference*.
- An online server setup and testing guide (SERVER.HLP, located in your IntraBuilder root directory after installation).
- The Professional Edition adds support (brokers) for CGI, NSAPI, and ISAPI and multiple IntraBuilder Agents for higher user loads. Also included: Netscape FastTrack Web Server, SQL Links for Borland InterBase,[a] and Microsoft SQL Server.
- The enterprise-scale Client/Server edition adds multiple IntraBuilder Agents on remote machines. Also included is the full set of SQL Links supporting the full range of industry-standard client/server SQL systems including Borland InterBase, Oracle, Sybase, Informix, IBM DB2, Microsoft SQL Server, as well as support for ODBC databases.

Supported interfaces and Web servers

- The Borland Personal Web Server
- NSAPI (Netscape FastTrack 2.0, Enterprise servers)
- ISAPI (Microsoft Internet Information Server)
- CGI (Common Gateway Interface, used by such servers as WebSite by O'Reilly & Associates)

Supported databases and data sources

- Microsoft Access (MDB) through ODBC
- Borland dBASE (DBF)
- Borland Paradox (DB)
- Borland InterBase*
- MS SQL Server*
- IBM DB2**
- Informix**
- Sybase**
- Oracle**
- Any 32-bit ODBC-supported data source**

* IntraBuilder Professional and Client/Server editions only.

** IntraBuilder Client/Server edition only.

Supported image formats

- Graphics Interchange Format (GIF)
- Joint Photographic Experts Group (JPG, JPEG)
- XBitmap (XBM)
- Windows bitmap (BMP)
- Device Independent Bitmap (DIB)
- Windows metafile (WMF)
- Enhanced Windows metafile (EMF)
- Tagged Image File (TIF, TIFF)
- PC Paintbrush (PCX)
- Encapsulated PostScript (EPS)

What you need to run IntraBuilder

IntraBuilder requirements are

- A personal computer with a 486DX or faster CPU
- Microsoft Windows 95 or Windows NT (3.51 or 4.0, Server or Workstation) operating system
- A CD-ROM drive (needed for installation only)
- 12MB RAM for Windows 95; 16MB RAM for Windows NT
- 30MB free space on your hard disk drive
- A VGA/SVGA monitor and graphics adapter
- A TCP/IP Internet or intranet connection
- A Windows 95 or Windows NT Web server application, such as the Borland Web Server, or, for the Professional and Client/Server editions of IntraBuilder, Netscape FastTrack, Microsoft Internet Information Server, or WebSite by O'Reilly & Associates.

Installing IntraBuilder

To install IntraBuilder,

1. Insert your IntraBuilder CD into your CD-ROM drive.

   (If you purchased IntraBuilder from the Borland Web site, follow the site instructions for download and decompression. When the setup files are decompressed, continue with the instructions below, replacing “d:” with the drive and directory that holds the decompressed file.)

2. If installing on Windows NT 4.0 or Windows 95, choose Run from your Start menu. If installing on Windows NT 3.51, choose Run from the File menu in Program Manager.

3. In the Run box, type

   ```
   d:setup
   ```

4. Click OK to start the installation program. Be sure to review the README.TXT file that appears at the end of the installation process. You should also carefully read and follow the directions that appear onscreen during all other stages of the installation.

5. After installing IntraBuilder, open the file SERVER.HLP by double-clicking on its icon in your IntraBuilder program group. This online Help file shows you how to configure your IntraBuilder-Web server connection, and provides step-by-step instructions for testing your connection using the pre-built business applications that are supplied with IntraBuilder.

Quick start notes

Use the instructions in the online Help file SERVER.HLP (located in your IntraBuilder root directory) to verify that your IntraBuilder-Web server connection is working properly. This Help file is listed as “IntraBuilder Server Help” in the IntraBuilder program group.

The test involves running the pre-built solution applications from a Web browser. If the pre-built solutions do not work, then neither will any applications that you create.

Once the connection is working properly, proceed to Quick tour, which introduces you to the design environment by creating a simple Web application.

Head start: SQL and JavaScript

Here are a few pieces of information you can use right away as you start developing and running your own applications. The online Help system offers many more pointers and procedures, including an extended tutorial covering all major program features.

Behind the scenes: The IntraBuilder architecture

How to connect your IntraBuilder application to an SQL server

Programming with JavaScript

How to connect your IntraBuilder application to an SQL server

To connect your IntraBuilder application to an SQL database, you need to configure your SQL Links Driver and BDE to access your SQL database. In this procedure you create an alias that BDE uses to locate the SQL database. You then add this alias to the Database object on your IntraBuilder form.

Consult the documentation for your SQL database management system product for specific guidance on the initial steps of the following general procedure (specific product requirements may differ).

Note The following instructions apply only to purchasers of the Professional and Client/Server editions of IntraBuilder. For the Professional edition, the instructions apply only to MS SQL Server and InterBase databases.

Procedure for connecting to SQL servers

1. Make sure you have properly installed the client software for the database management system product to which you want to connect (Oracle, Sybase, Informix, InterBase, or MS SQL Server).
2. Define server names or other connection strings in the product's required configuration files. For example, in Oracle, TNSNAMES.ORA, or in Sybase, SQL.INI, and so on.
3. Test the connection by using the database vendor's connection utility (such as Sybase's SYBPING.EXE). If you cannot "ping" the server with this utility, BDE and IntraBuilder will probably not be able to access it either.
4. Make sure that both BDE and the Borland SQL Links products are properly installed. These core products are included with IntraBuilder Professional. If properly installed, the SQL Links drivers for Oracle, Sybase, MSSQL, Informix, and InterBase appear on the Drivers page of the BDE Configuration Utility (BDECFG32.EXE, found in your IntraBuilder/BDE directory).
5. Run BDECFG32.EXE and add an alias for the SQL server. Settings for the alias may vary according to vendor. For more information on how to set up an alias, consult the online BDECFG Help file while using the utility.
6. In IntraBuilder, open the IntraBuilder Explorer, click the Tables tab, then choose the SQL server alias from the dropdown menu in the Look In box (at the top of the IntraBuilder Explorer). You are then prompted for a login name and password to connect to that SQL server database. Once you connected successfully, you will see the tables in that database in the IntraBuilder Explorer.

The easiest way to use a table in a SQL server database in a form or report is to drag the table from the IntraBuilder Explorer onto the surface of the form or report in the Form or Report Designer. This automatically creates the Database object required to connect to the database, and the Query object for that table.

IntraBuilder often offers alternative ways of doing things. You can also create the Database object in a script or drag a database component from the Component Palette to the design surface and set its databaseName property to the alias you created in BDECFG32.EXE.

Programming with JavaScript

As mentioned earlier, the programming language used in IntraBuilder is extended JavaScript. It is used to create all forms, reports, home pages, and objects in every part of the program; every time you draw a control on a form or report, the result is immediately described in JavaScript and saved to the underlying program.

You can examine the JavaScript source for any saved program: right-click on any JFM or JRP file in the IntraBuilder Explorer and choose Edit As Script.

As you'll discover if you're new to it, the JavaScript language is a relatively easy language to learn and use (relative to other popular Web development languages like Java and Perl). Developed at Netscape and supported by Microsoft's ActiveScript initiative, JavaScript has gained widespread popularity for its versatility and flexibility.

It has much in common with other popular scripting languages, and such features as a dynamic object model and automatic data-type conversion have made it the scripting language of choice on the Internet.

IntraBuilder's JavaScript extensions add the kind of power and sophistication needed by database developers—including literal arrays, codeblocks, and exception handling. IntraBuilder also adds several constructs found in Java and a range of new classes for database management. Server-side JavaScript also provides statements for object inheritance and subclassing—always a must for serious developers.

IntraBuilder's JavaScript language is described in detail, with plenty of sample code, in both the online Developer's Guide and the online Language Reference.

To open either of these references, choose Help Topics from the IntraBuilder Help menu.

Behind the scenes: The IntraBuilder architecture

IntraBuilder is made up of several components, each of which has a distinct but equally important purpose.

- The IntraBuilder Designer is the design and development environment in which you create and maintain your applications.
- The IntraBuilder Server is made up of the IntraBuilder Broker (INTRASRV.DLL and INTRASRV.ISV) and one or more IntraBuilder Agents (instances of INTRA.EXE). The IntraBuilder Server is surfaced through the Agents, which appear as minimized icons on your desktop. You configure additional agents to handle increased user loads. IntraBuilder Server dynamically converts IntraBuilder JavaScript applications into HTML pages on the network.

The process looks like this:

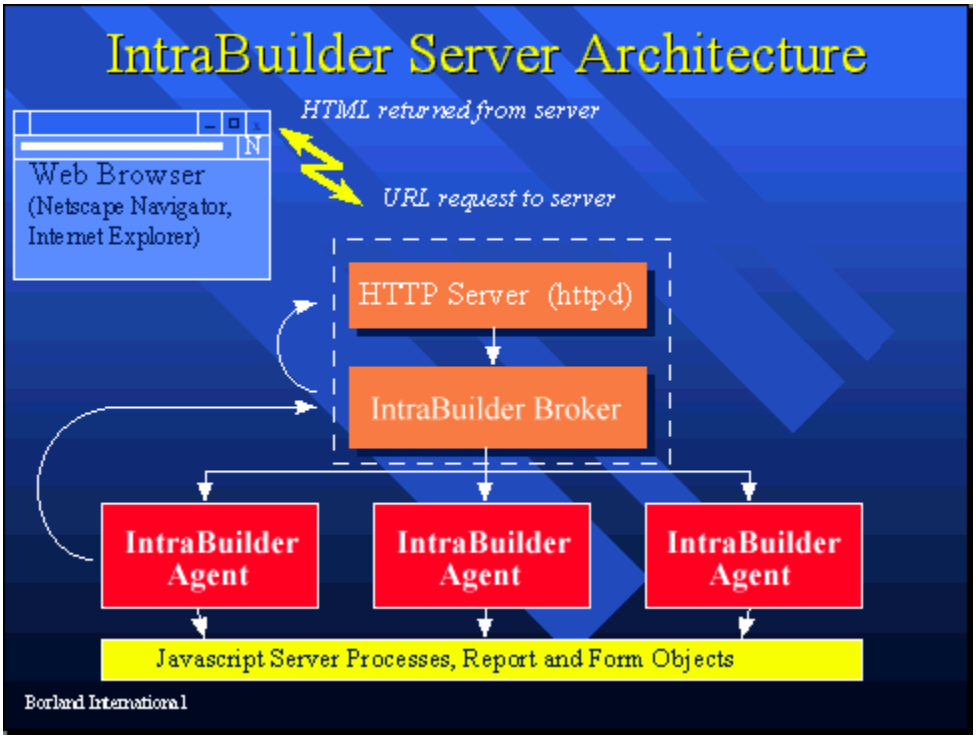


Figure 1.1 IntraBuilder Server architecture

Getting Help

Related topics

The IntraBuilder online Help system is a comprehensive reference to all aspects of the application. In addition to the information in your printed documentation, the Help system offers information that wasn't available at or was changed after press time.

You can also add to your IntraBuilder Help system by obtaining addenda and updates from the IntraBuilder home site. For information on this service, see the IntraBuilder documentation page at http://www.borland.com/techpubs/intrabuilder.

If you're unfamiliar with Windows Help, this section of *Getting Started* shows you how to use the various features of the IntraBuilder system to full advantage.

Help books

Related topics

The main Help system is subdivided into five sections:

- *Getting Started* (an online copy of this booklet)
- The Server Setup and Troubleshooting Guide, a detailed reference to setting up your Web server to work with IntraBuilder and step-by-step guide to testing your IntraBuilder Server connections
- The complete *Developer's Guide*, which includes a detailed tutorial covering all major IntraBuilder features
- A complete *Language Reference*, with code samples you can cut and paste directly from the Help window

Separate Help files are also provided for the included Borland Web Server and Borland Database Engine Configuration utility. For a view of the Help system topic structure, choose Help Index.

Context-sensitive Help

Related topics

IntraBuilder provides three levels of context-sensitive Help:

- F1 Help, which opens a pop-up or full Help topic that describes the current control or language element when you press the F1 key. F1 Help is available for menu items (including right-click menus); most controls and elements within windows, dialog boxes, toolbars, and property inspectors; and in any editing window for a highlighted word or phrase.

  Note Not all phrases are indexed; generally, only language elements and control names are documented this way.

- Status bar Help, which appears on the panel at the bottom of your main IntraBuilder window. These descriptive captions appear as you move through menus (including right-click menus) or pass your cursor over toolbar buttons.
- “Flyover” Help, which offers a pop-up description of toolbar buttons when you let your cursor pause over a button.

General and procedural Help

Related topics

- Most IntraBuilder dialog boxes offer a Help button that opens a full-topic description of the dialog box and the controls it contains. These topics often provide additional links to usage tips and procedural topics.
- The Help system contains a number of procedural topics to guide you through common tasks. You can find a list of these topics in the How To section of the Help Contents window. You can also get the list by choosing *How To* in the Help Index (see the next topic for an overview of Help's indexing and Find features).
- In addition to the single-task procedures, the system features a comprehensive tutorial that helps you learn all of the major features of IntraBuilder in a step-by-step guided tour. (Help Index key word: ***Tutorial***.)
- The IntraBuilder *Language Reference* is a master reference of IntraBuilder's extended JavaScript language implementation. Its main topics, including several cross-reference indexes, are available in the Help Contents window. You can also get Help on language elements by highlighting a keyword and pressing F1 while in any editor.
- Other user interface and programming topics are covered in depth in the *Developer's Guide* section of the system. For section overviews, see the Help Contents window.

Help Contents, Index, and Find features

Related topics

When you choose Help Index from the IntraBuilder Help menu (or if you open the Help file directly from your Windows Explorer or File Manager), the Help Contents tabbed dialog box appears.

The dialog box contains three tabbed sheets:

- Contents is the general Table of Contents for the Help system. To expand the list, double-click on book icons; to open topics, double-click on page icons.
- The Index tab displays a list of key words and phrases associated with topics. Some words and phrases are associated with a number of topics; when you choose one of these (by double-clicking on it), a new list appears with a list of topics to choose from.
- The Find tab provides a full-text search database. You can customize the search database by choosing Customize Search Capabilities the first time you choose the Find tab. You can also customize these capabilities when you generate a new search database. To remove an existing search database, delete the *.FTS file that's in the same location and has the same base name as the Help file.

Help topic window controls and navigation

Related topics

The following are the general rules of navigation for IntraBuilder Help topics. For more on Windows Help, choose How to Use Help from the IntraBuilder Help menu or open the file WINLP32.HLP in your Windows/Help or WinNT/Help directory.

To view related topics or examples, choose the links below the title in a topic window (not all topics have these links).

Links within a topic body either let you jump to other topics (if the link has a solid underline below it) or pop up a definition or list (if the link has a dotted underline).

Other navigation controls:

- Browse buttons (<< and >>), where enabled, take you to the next or previous topic in a series of related topics.
- The Tracker button opens a small History list in a separate window that lets you return to recently viewed topics in the current Help system. Note that this list applies only to the current Help session. When you close Help, the History list is cleared.
- The Back button returns you to previously viewed topics in reverse sequential order.

Annotations and bookmarks

Related topics

To add notes to IntraBuilder Help topics, choose Annotate from the topic window Edit menu or right-click in the topic window and choose Annotate from the pop-up menu.

When you type a note and press Save to save it, a small paper-clip icon appears in the top left corner of the topic window. To read or edit a note, click the paper-clip icon.

To define bookmarks in your IntraBuilder Help file, click Define from the Bookmark menu. You can then type in an identifier for the current topic. The identifier is added to the Bookmark menu. You can then return to the topic any time by choosing the topic identifier from the Bookmark menu.

Warning! If you overwrite an IntraBuilder Help file with an updated version of the Help file, any annotations and bookmarks you created in the older Help file won't show up in the new file. A workaround to this Windows Help limitation is to rename the older Help file before copying the new version to your IntraBuilder/Bin directory. After that, immediately go to your Windows/Help or WinNT/Help directory and rename the appropriate *.ANN and *.BMK files to match the base file name you gave to your older Help file. You can then access your old Help file notes and bookmarks by opening the old file directly from disk.

Copying Help text

Related topics

You can copy text from a Help topic or pop-up and paste it into the Script Editor, Method Editor, or Script Pad (or any other place in Windows in which you can paste from the clipboard). This allows you to paste a function syntax or block of JavaScript from your script.

To copy text from a pop-up, right-click anywhere in the pop-up menu and choose Copy from the pop-up menu. The entire text of the pop-up is copied to the clipboard (you can't select text in a pop-up).

To copy text from a topic window, first select the text you want to copy, then either use the right-click pop-up menu or choose Copy from the topic window Edit menu.

In either a pop-up window or a topic window, you can also use the keyboard shortcut Ctrl+C to copy text.

Printing Help topics or pop-ups

Related topics

You can print any Help topic or pop-up. To print a topic, choose Print from the topic window File menu or right-click in the topic window and choose Print Topic from the pop-up menu. To print a pop-up topic, right-click on it and choose Print Topic from the pop-up menu.

You can also print a topic or a range of topics from the Help Contents dialog box. To print all topics in an online book, choose the book icon that contains the series of topics you want to print and click the Print button. To print a single topic, choose its page icon and click Print.

More on using Help

Related topics

For more information on using a Windows Help system, choose How to Use Help from the IntraBuilder Help menu or open the file WINHELP.HLP in your Windows/Help or WinNT/Help directory.

Removing IntraBuilder

- Windows NT 3.51: Click the Uninstall icon in your IntraBuilder program group in Program Manager.
- Windows NT 4.0 or Windows 95: Use the Add/Remove Programs applet in your Windows Control Panel.

Note During uninstallation, you also have the option of keeping any shared program libraries on your disk that may be needed by other programs. Even if you choose to remove the shared files, other files and directories may remain on your disk after uninstallation. These remaining files are usually forms, applications, directories or other items you created while using IntraBuilder.

For other issues that may affect IntraBuilder removal, see INSTALL.TXT and README.TXT. Both files are located on your IntraBuilder CD and installed into your IntraBuilder root directory.

Documentation updates and additional information resources

Related topics

The IntraBuilder home site (http://www.borland.com/intrabuilder) helps you find the most current information about IntraBuilder and JavaScript. The IntraBuilder documentation page (http://www.borland.com/techpubs/intrabuilder) lets you download online Help updates and read or download technical notes, tips, and other materials that will further your understanding of the program.

Online glossary

A searchable version of the Dictionary of PC Hardware and Data Communications Terms, by Mitchell Shnier and published by O'Reilly & Associates, is available online at:

`http://www.ora.com/reference/dictionary/tsearch.cgi`

Quick tour introduction

This section of the Help file introduces you to the basics of using IntraBuilder. The tutorial takes about 20 minutes. You'll create an intranet application called "Fonelist," consisting of a table, a report, and a Web page form. The form that will appear on users' browsers will contain fields and various buttons that demonstrate the essential functionality of IntraBuilder forms.

Deployed on your intranet, this simple form can collect names and home phone numbers for those in your organization. Users can view the form with a standard Web browser, regardless of platform or location, browse through the records, and post a new record to the table, thereby populating the table for you. Users can also search for a particular record and view a report that lists all names and phone numbers in the table.

Creating and deploying an IntraBuilder application

IntraBuilder applications can consist of tables, forms, reports, and home pages or other files, including graphics, query files, and scripts to automate certain tasks. You can create IntraBuilder applications by using the experts, the designers, or both.

The Script Pad and Method Editor are available so you can view, test, create, and edit the underlying JavaScript elements generated by the experts and designer interfaces. To see the entire script of your applications, use the Script Editor. However, it isn't necessary to work with code; you can create working applications entirely by using experts and the visual designers, as this tutorial demonstrates.

Here is the general procedure for creating and deploying an application:

1. Create a table (or identify an existing table whose data you want to use).
2. Create a form and link it to the table.
3. Enhance the new form in the Form Designer (optional).
4. Make the table and form files accessible to IntraBuilder Server.
5. Start IntraBuilder Server and your chosen Web server.
6. Use your Web browser to access the form so you can view and post data.

This tutorial follows the general procedure, including modifying a form in the Form Designer.

To satisfy step 4, you will develop and run your application in the same place, on the same machine; this makes the files you develop in the IntraBuilder Designer automatically accessible to the IntraBuilder Server. In addition to the IntraBuilder Designer, you must be able to run the IntraBuilder Server, your Web server, and your Web browser all at the same time (although you can quit the IntraBuilder Designer if you're low on memory when running your application through the server).

By default, the IntraBuilder Server looks for its files below the IntraBuilder root directory (C:\Program Files\Borland\IntraBuilder, if you installed to the default location). You will place the Fonelist files you create in a subdirectory of the IntraBuilder root directory.

Step one: Setting up

To begin the tutorial,

1. Start the IntraBuilder Designer. You can double-click the IntraBuilder Designer icon, or choose IntraBuilder Designer from the Start Menu.
2. Create a directory under the IntraBuilder directory, and name it Quick. If you installed to the default location, the full path name to the new directory will be C:\Program Files\Borland\IntraBuilder\Quick. You can either make the directory in your Windows Explorer (or File Manager if using NT 3.51), or you can remain in IntraBuilder, choose Script Pad from the View menu, and type this single command into the Script Pad:

   ```
   _sys.os.makeDir("c:\\program files\\borland\\intrabuilder\\quick")
   ```

   Then press Enter.
3. If the IntraBuilder Explorer is not already open, open it by choosing View|IntraBuilder Explorer.
4. Using the Folder button and the Look In box, locate the project directory that you just created: C:\Program Files\Borland\IntraBuilder\Quick. This makes it IntraBuilder's current directory, which will be used to store the application files created in this tutorial.

Step two: Create a table

Although you might frequently connect IntraBuilder to existing databases, for purposes of demonstration this tutorial starts by showing you how to create a new table in IntraBuilder.

1 In the IntraBuilder Explorer, select the Tables tab.

The IntraBuilder Explorer is an organizing tool for managing files you create with IntraBuilder. If you have configured remote SQL servers in BDE, their aliases will appear when you click the drop-down arrow beside the Look In box.

2 Double-click the Untitled icon. Or, drag the icon onto the desktop. Or, choose File|New|Table.

Any of these actions opens the New Table dialog box, where you have the choice of using either a designer or expert to create your new table.

3 Click Expert. The Table Expert appears.

In the Sample Tables list, you see a list of sample tables provided with IntraBuilder. You can choose any of these tables to use as a template for your new table.

4 Click the Personal Info table from the list of sample tables. The fields in the Personal Info table now appear in the From Sample Table list.

5 Double-click, in this order:

FIRST_NAME
LAST_NAME
HOME_PHONE
NOTES

These fields now appear in the For New Table list. (Copied fields include all properties from the original table.)

Note You don't have to stick with the fields from just one table. You can select other tables and choose fields from them, as well.

6 Click the Next button. The second step of the Table Expert appears.

Here you can specify a table type. For this tutorial, leave it as dBASE (it will have a DBF extension).

Note The default list always offers the Standard table formats, dBASE and Paradox. The Professional edition of IntraBuilder offers the Microsoft SQL Server and Interbase formats. The Client/Server edition of IntraBuilder offers additional formats, such as Oracle, Sybase, DB2, or Informix. What formats you can use depends on the databases you have configured in the BDE.

4 Click Run Table. The Save Table dialog box appears.

5 Name the new table FONELIST.DBF and save it in your new Quick directory.

The table now appears in the Table window in a view that shows one record at a time. You can see that FONELIST.DBF contains the four fields you copied from the Personal Info sample table. Now you have a table on which to base a Web form.To try it out, add a record (row) to your new table:

1 Right-click the table window and choose Add Row.

2 Type in your name, phone number, and a note.

3 Right-click the table again, and choose Save Row. You have added your first record to the Fonelist table.

4 Close the table window.

Now you've finished creating a table for your IntraBuilder form to use. The table is empty (except for the test record you just entered), but the structure you need is there. The IntraBuilder Explorer lists your new table on the Tables page.

Step three: Create a form

Now you'll create the data-entry form that will appear as a Web page on the users' browsers, allowing them to browse, edit, and post records to your new table.

1. In the IntraBuilder Explorer, click the Forms tab and double-click the Untitled icon (the one that looks like the most complete form) or choose File|New|Form.
2. Click Expert. The first step of the Form Expert appears, where you can link a table to your new form.

   The Form Expert shows the tables in the current directory. Because you saved the table FONELIST.DBF in the current directory, it appears in the list. If it is the only table there, FONELIST.DBF appears already selected in the Selected Table Or Query box.
3. Make sure FONELIST.DBF is selected and click the Next button.

   Step 2 of the Form Expert appears. It shows the table's fields and asks you to pick the fields you want to display in the form.
4. Click the topmost double-arrow to copy all the fields to the new form.

   All four fields in the source table now appear in the Selected list box.
5. Click Next, and step 3 appears, asking you to choose a layout style: either columnar or form.
6. Select Columnar Layout, and click Next.

   Step 4 asks you to choose a predefined visual scheme or create your own look by choosing fonts, font colors, and background colors.

Note When you're creating your own scheme, used the tabbed section of this step. On the Title page, set the font and color of the form's title. On the Labels page, set the font for the field and object labels. On the Form page, set background and foreground colors of the HTML form page. A group of settings in all three can be saved in the Scheme box. You can also change the color and font scheme later by choosing Layout|Set Scheme in Form Designer, which displays a similar dialog box.

7. In the Scheme box, try selecting a few of the predefined schemes and preview the results in the Sample pane at the upper left corner. Then select the IntraBuilder Default scheme, and click Next.

   In step 5, the expert asks you to select buttons to enable users to navigate, update, query, or filter the table, and to jump to another form or view a report. The functionality of these buttons is preset. You can select them individually or click the All button to add all of them. To find out more about these controls, click the Help button in the expert.
8. Select the Buttons control type and set Location On Form to Bottom. This generates standard HTML buttons in a row at the bottom of the form.
9. Check the Next, Previous, Add, and Query By Form buttons. These four buttons enable users to browse forward and backward through the rows (records), post their names and phone numbers to the table, and search for the phone numbers of others.
10. Click the Next button.

    Step 6 appears, giving you the choice of running the new form immediately or modifying it in the Form Designer.
11. Click Run Form.

    The Save Form dialog box appears.
12. Name your new table FONELIST.JFM and save it in the Quick directory where you saved the associated table, FONELIST.DBF. If you haven't changed directories, the Quick directory is still your current directory.

    And there it is—your new form.

It shows the four fields you chose to display from the associated table (FONELIST.DBF) and the buttons for navigating the records, posting new records, and running a query on the rowset.

This is what users will see on their browsers when they run the form through the Web server.

This is a fully functional form, running locally within the IntraBuilder Designer.

You can now test the functionality of your new application by creating a few records through the form locally and trying the navigation buttons to move back and forth through them. If you created a test record when you first created the table, you should be able to view and edit that record now through the form. Once you have a few records in the table, try the Query feature.

1. Click Add. This puts the table into Append mode, which clears the data entry fields in the form, allowing entry of a new row of data to the table.
2. Enter first name, last name, phone number, and notes in the text boxes. Note that you can tab from field to field.
3. Click Add. This posts the first record to the table FONELIST.DBF and puts the table back in Append mode.
4. Type in data for another person.
5. Click Add.
6. Add a few more records.
7. Now you have some records in the table. Try clicking the Previous and Next buttons to navigate between the two records.
8. Now click the New Query button. The fields clear, leaving a blank form, and the button name changes to Run Query.
9. In the First Name field, type a first name as it appears in one of your records, then click the Run Query button.
10. The form displays the full record with that first name. Click the New Query button again, and this time specify a last name.

At this point you could deploy your new table and form, the Fonelist application, over the Web and immediately see what it looks like. If you want to try deployment right away, you can skip to Step five: View the finished form in your Web browser. Then come back and go through the next topic, which demonstrates how to make a few quick enhancements to the form by using the Form Designer.

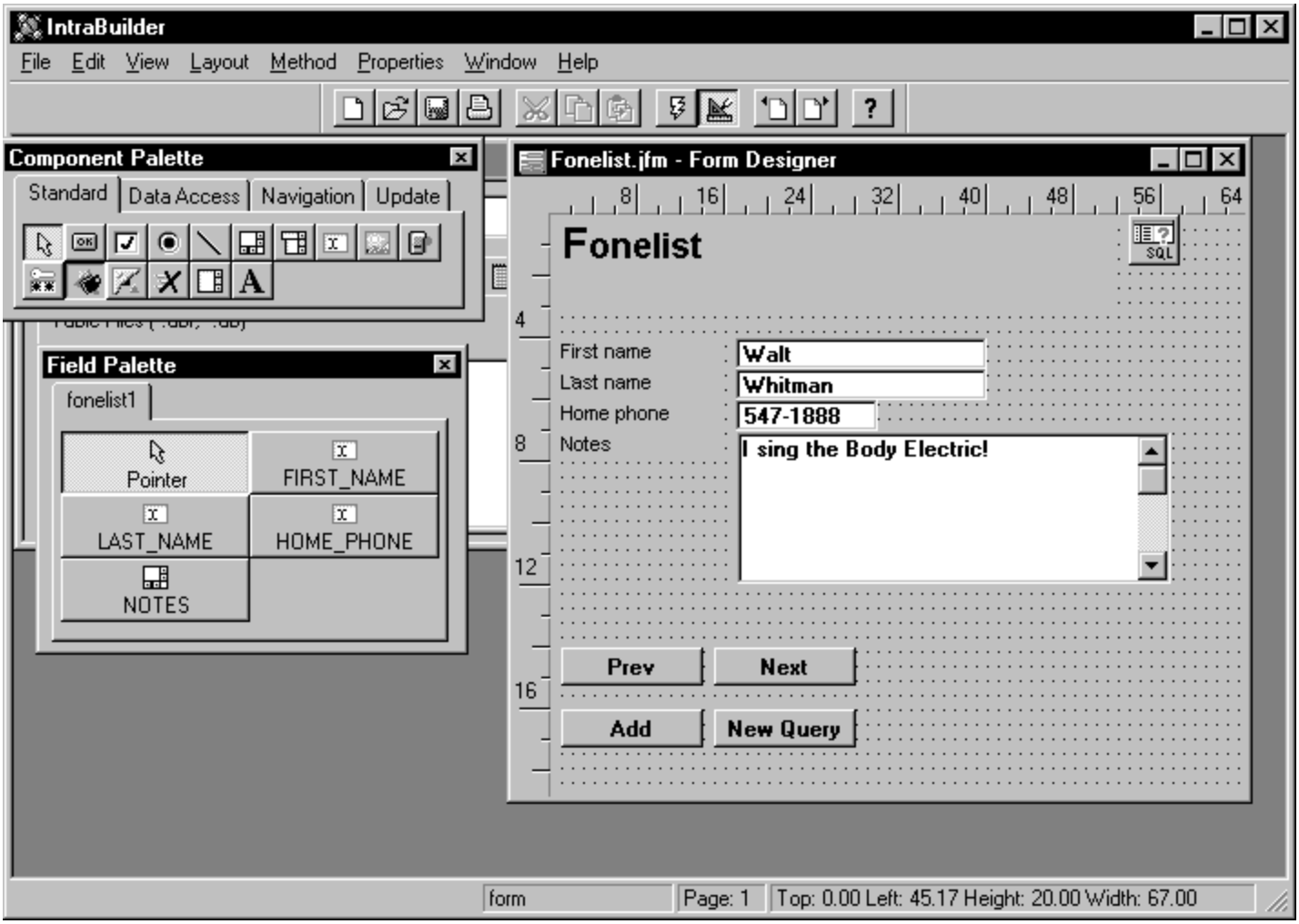
As a compact alternative to HTML buttons, you could select the Images control type, which creates icon controls. If you check all four checkboxes in the Navigate group, the Form Expert creates a single icon containing all four controls.

Update and Search Or Limit buttons can be used together or individually as icons. Also, you can add HTML links to other forms and to reports by specifying them at the bottom of the page. You can change anything later in the Form Designer.

Step four: Enhance the new form in Form Designer

While using the Form Expert is a quick and easy way to put together a working form, often you will want to modify the form's appearance or functionality by using the Form Designer. Here, as an introduction to the Form Designer, you'll add a graphic image to the Fonelist form, change the title, rearrange the fields, and add a button that's linked to a simple report you'll create.

1. With the Fonelist form still open in Run mode on the IntraBuilder desktop, click the Form Design button on the toolbar. This places IntraBuilder in Design mode, and displays the design tools that comprise Form Designer.
2. If the Field Palette and Component Palette aren't already displayed, choose View|Field Palette and View|Component Palette just to see what they look like. This tutorial doesn't use them, but when you do want to place a field or component on a form, just drag and drop from the palette to the form. Your desktop in Form Designer should look like this:



Notice the small object in the upper right, labeled "SQL." That is a Query object, the JavaScript object linking this form to the table, FONELIST.DBF. Ignore it for now; it will not be visible to users when the form is running.

Add a graphic image (Step 4, cont'd)

The first enhancement you'll make to your form is to add a graphic image.

1. Use Windows Explorer to find a graphic image you'd like to place on your form, and copy it to your Quick directory. (There are some sample images in the Clipart subdirectory under the IntraBuilder root directory.) IntraBuilder accepts a wide range of image formats.
2. Return to IntraBuilder and click on the IntraBuilder Explorer to bring it into focus.
3. Click the Images tab to see the graphic image file you just added to the Quick directory.

Note You could also have used the IntraBuilder Explorer to locate an image file; when you do it this way, you have to add the path to the image file name in the Inspector. Putting a copy of the image file in your Quick directory is simpler for this tutorial.

4. If you want to view the image, just drag the graphic file out onto the IntraBuilder desktop. The image is displayed in the IntraBuilder Image Viewer. You can right-click on the picture to display a menu that lets you paste, import, export, and set the Image Viewer properties.
5. From the Images page of the IntraBuilder Explorer, drag the image's icon or file name to your form surface, and drop it there.

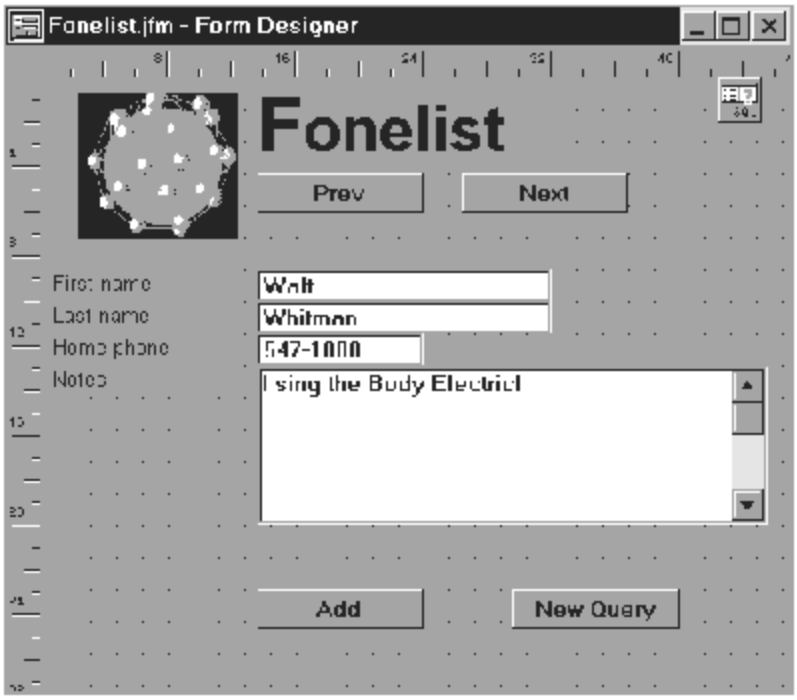
   The image appears on the form. If it's not what you expected, you may have to delete it and then open the file in a graphics application like Paint, modify it, save it, and then drag the file name onto the form again.

Rearrange form objects (Step 4, cont'd)

You can drag the picture to any place on the form. In Form Designer all objects can easily be moved around by selecting and dragging them. When you select an object, whether button or field or graphic, it is surrounded by handles, so you can resize or move it.

1 Try rearranging the objects to create a different layout. Notice how objects snap to the grid, coming into perfect alignment. (You can set a finer grid; to do so, choose Properties|Form Designer Properties while in the designer.) The Form Designer also provides many alignment tools in the toolbar.

After moving objects around, the Fonelist form might look like this:



Change the form title (Step 4, cont'd)

You have arranged the form more to your liking. But what about fixing that silly title? Easy.

1. Select the title, "Fonelist" so that the handles appear.
2. Choose View|Inspector or press F11 (or, if the Inspector is already on the desktop, just click it to bring it into focus).
3. Make sure the Properties tab is selected, and then select the text property. To the right of the text property box are two tools. Choose the wrench tool.

   The Text Property Builder appears. In the Text Property Builder, you can add or revise text and add HTML font styles. The URL Tag, Color Tag, and Custom Tags panels let you set links to other topics or websites, choose colors, and code HTML directly on the selected text.
4. In the Text Without Tags panel (the upper right box), type Home Phone Directory.
5. If you want to modify the style of the text, select it in the Text Without Tags panel, and then try the other tools in the Text Property Builder. If you need help, click the Help button in this dialog box.
6. When you have finished, click OK.

   The Fonelist form appears on the Form Designer surface with your new title. To accommodate the larger title, you might have to move buttons and resize the Title object (by selecting it and then dragging on one of the handles in the direction you want to enlarge it.)

Add access to a report (Step 4, cont'd)

You might want to offer your users a report that displays the entire contents of FONELIST.DBF that they can view or print from their browsers.

Create a simple report

Creating a report is a quick process with the Report Expert:

1. From the Reports page of IntraBuilder Explorer, double-click Untitled.
2. Choose Expert.
3. In step 1, select Fonelist, and click Next.
4. In step 2, accept the default.
5. In step 3, select all fields.
6. In step 4, select LAST NAME for sorting.
7. In step 5, just click Next (no summaries).
8. In step 6, accept the default.
9. Run the report. Name it FONELIST.JRP.
10. Close the report preview window.

Add a link to the report (Step 4, cont'd)

To add a link to a report—or to another form—is as easy as finding the file name in the IntraBuilder Explorer and dragging it to the form in the Form Designer:

1. With the IntraBuilder Explorer still set to the Quick directory, click the Reports tab. You should see FONELIST.JRP listed on the Reports page.
2. Drag the icon or file name to the visual design surface of the Fonelist form, and drop it there. It appears on the form as an active button. When the user clicks the button in the browser, the Fonelist report will be displayed.
3. Click the Run button in the toolbar to test your finished form.
4. Close the form.

The Inspector lets you edit the properties, events, and methods of JavaScript objects.

Step five: View the finished form in your Web browser

To view the form on a Web browser, you must start both the IntraBuilder Server and the Web server. If not already started, do so now. (If you are low on memory, you can quit the IntraBuilder Designer.)

Start your Web browser. In the location box, type the URL of your new form, such as:

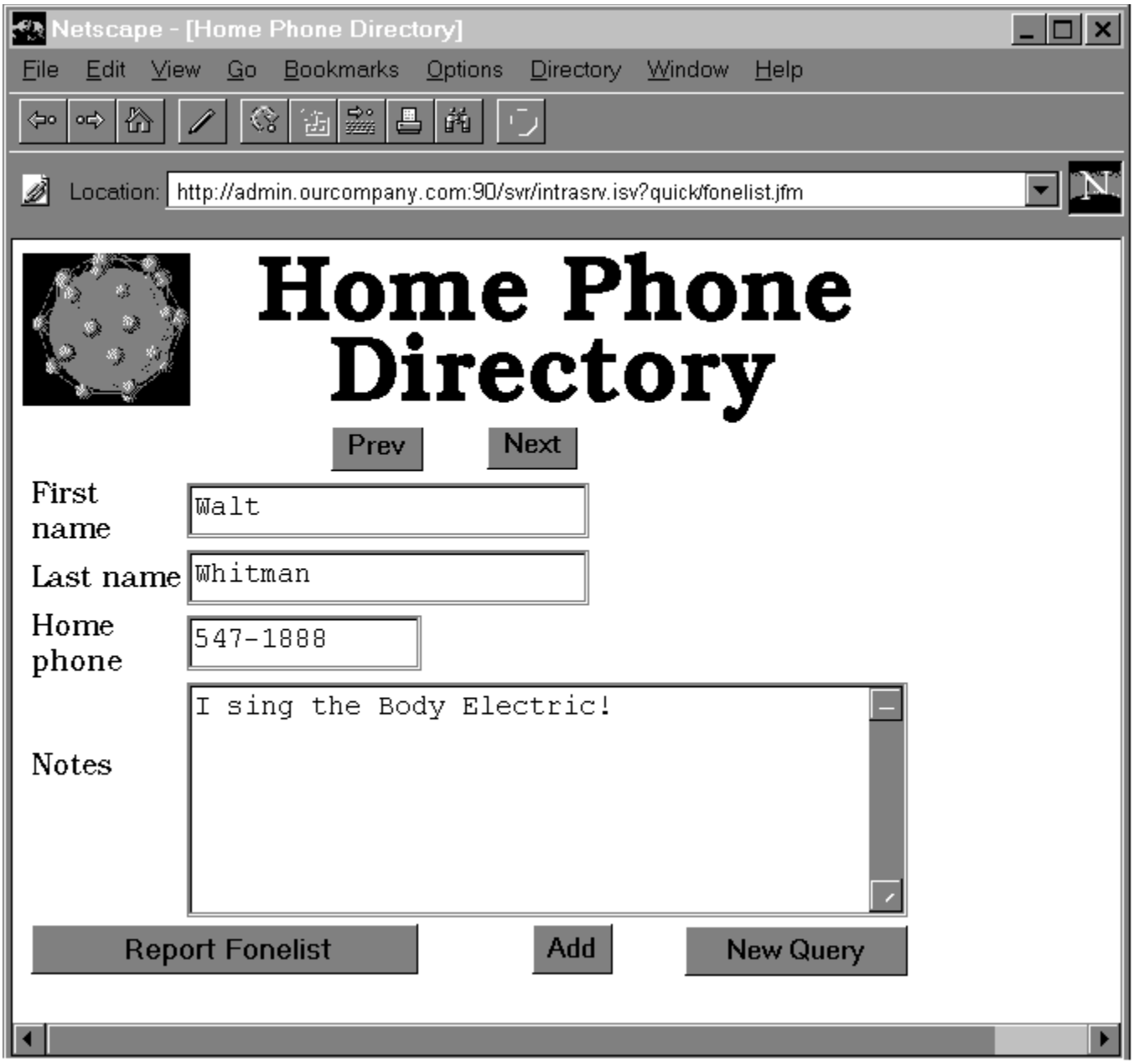
```
http://admin.ourcompany.com:90/svr/intrasrv.isv?quick/fonelist.jfm
```

The first part of this URL depends on the installation and configuration of your Web server and may include a port number for the server computer, in the above case, port 90. The /svr/ part is the suggested alias for the directory where INTRASRV.ISV is located; this could vary at your installation. The last component of the URL should be a form of “intrasrv.isv?quick/fonelist.jfm”, which tells IntraBuilder to open the Fonelist form in the Quick subdirectory.

Because you are testing the form through a browser on the same machine, you can also use the “localhost” address (as long as your TCP/IP host configuration is correct), as in:

```
http://localhost/svr/intrasrv.isv?quick/fonelist.jfm
```

Here is how FONELIST.JFM looks in Netscape Navigator:



Try it out in the browser. Remember to first click Add to put the form in Append mode. Then enter a first name, last name, a home phone number, and notes. Click the Add button again. Add several records. Then use the Next and Previous buttons to browse through the records you have created. Try the Query button, type a name in either the First Name or Last Name fields and see if the correct record is displayed. Click the report button to view a complete list of all the names, phone numbers, and notes in the table.

Users can access this simple form from anywhere in the world by using a standard Web browser. They can also query the table, enter data, edit it, post it to the database, and browse through the other records.

Now you've learned the basic procedure for creating tables, forms, and reports by using the experts. The Developer's Guide shows you how to use IntraBuilder Designer to create tables, forms, reports, queries, and home pages from scratch or to modify what an expert created for you. See Part II, "IntraBuilder JavaScripting," to learn about JavaScript classes and objects and how to create more complex, sophisticated Internet applications.

Using the online Developer's Guide

Related topics

This online Developer's Guide shows you how to use IntraBuilder to create database applications that you can deploy over the Web.

Part I, "Working in the visual designers," describes the procedures for creating IntraBuilder tables, forms, reports, and home pages. It also covers security and deploying your finished application over the Web.

Part II, "IntraBuilder JavaScripting," explains how to use IntraBuilder's extended JavaScript to give your applications additional functionality. Numerous real-world application requirements and enhancements are illustrated with JavaScript examples. You don't need Part II to operate IntraBuilder or to create and deploy IntraBuilder applications.

Part I, IntraBuilder basics

Related topics

After you have followed the instructions in Help's *Getting Started* section to install and test IntraBuilder Server, and gone through the brief tutorial, "Quick Tour," you are ready to begin creating a Web application.

Part I presents IntraBuilder's design and development tools. If you are familiar with visual RAD environments, you can probably rely on the onscreen Windows Help or occasionally refer to the Help sections on the Table Designer, Form Designer, Report Designer, Visual Query Builder, and Home Page Expert.

Table Designer describes IntraBuilder's simple Table Designer, an easy way to build or modify tables right in IntraBuilder. (Of course, you can also connect your IntraBuilder forms to Standard and SQL databases throughout your enterprise. For that, see Table Designer.) This chapter also explains how to set up referential integrity for tables that support it. If you are using existing tables or databases, you can skip this section.

Form Designerexplains how to use the many tools and options of the Form Designer. The Form Designer tools provide extensible component palettes and full access to the properties, events, and methods of all form objects. All the major RAD tools are described, along with basic IntraBuilder file operations, and customization of the interface.

Report Designer describes the tools you use to create and format reports for printing or as an alternative way to deploy information on the Web. Many Report Designer tools and operations are also available in the Form Designer .

Visual Query Builder explains how to use the Visual Query Builder to construct complex SQL statements. Although you can build basic queries using simple tools available in the designers, the Visual Query Builder gives you many additional options.

Home Page Form shows how to use IntraBuilder's Home Page Form Expert to quickly create an IntraBuilder form that generates a Web-style home page as a central focus for an application, with links to associated forms and reports. You can customize IntraBuilder home pages by using the Form Designer.

Security gives details on IntraBuilder's security features, including automatic password verification service, table- and field-level security, and custom login forms for application-level security.

Part II, IntraBuilder JavaScripting

Related topics

This section of the online Developer's Guide takes you through the process of building a full-fledged Internet application, called the Threaded Message Database (TMD). TMD is a groupware tool for tracking e-mail discussion threads. Special design requirements are considered, to give you a deeper understanding of the JavaScript behavior underlying IntraBuilder applications.

Introduction to IntraBuilder programming introduces the TMD project that provides the illustrative foundation for the discussions in the succeeding chapters.

Accessing tables discusses the properties and behaviors of data access objects, including Query, Rowset, Field, Database, and Session objects. Also provided is a procedure for connecting to remote SQL databases via the BDE and SQL Links (relevant to IntraBuilder Professional and IntraBuilder Client/Server editions only).

JavaScript forms discusses the JavaScript form file format. In this chapter you build the form for the TMD project.

Database access from forms studies Data Access objects in detail and analyzes login and security considerations. In this chapter you create the table for the TMD project.

Customizing the application takes you through the customization of the TMD project, in particular creating a custom login form and handling subtleties of user feedback and user interface polish.

Custom forms and components is an in-depth explanation of how to use the properties, events, and methods of form components.

Integrating reports guides you through the creation of a report for the TMD project, showing how to integrate reports with applications. This completes the TMD project.

Client-side JavaScript shows how to create JavaScript applets you can embed in forms to execute on JavaScript-enabled browsers.

Introduction to IntraBuilder programming

Related topics

IntraBuilder's rapid application development (RAD) capabilities are built on a solid, object-oriented foundation. IntraBuilder reads and writes an enhanced version of JavaScript, the industry-standard Internet scripting language.

While you can create viable working applications without touching a line of code, the extended JavaScript foundation is readily accessible if you need—or want—to further customize your Web applications and include advanced features. You will find that IntraBuilder embodies an unmatched synergy between databases and the Web that empowers you to create multi-tiered solutions with multi-dimensional application partitioning.

The challenge of Web database applications

Related topics

The World Wide Web is a compelling new platform that enables anyone with a Web browser to access your data from anywhere without having to go through the trouble of downloading your application and installing it. By running powerful database applications through a browser, users everywhere can get instant access to your data.

World Wide Web basics

Related topics

A Web browser and Web server are needed for any type of interaction over the World Wide Web. Anyone who has surfed the Web knows that the Web browser asks for a page by specifying a URL (uniform resource locator). The URL contains the name of the Web server and the name of the requested page on that server. The Web server gets that page, which is written in HTML (hypertext markup language), and sends it to the Web browser. The browser then renders, or displays the page, as best as it can, ignoring any HTML elements it does not support.

A basic page contains formatted text and graphics, either of which can be linked to other pages. Links contain URLs; clicking on them displays the specified page, as if you had typed in the URL into the Web browser yourself.

The HTML 2.0 specification introduced HTML forms. These forms contain familiar data-entry elements, such as text areas to enter information, check boxes to select options, and radio buttons for multiple-choice questions. When you have finished entering data or selecting radio buttons and check boxes, you click a button; this sends the contents of the form to the Web server. The Web server passes those values to another program, usually through some form of a protocol named CGI (common gateway interface). These external programs, typically written in a language like Perl or C, may modify the form values (for example, post online survey data to a table or use the values in a query) and then formulate a response in HTML. This response is then passed back to the Web server, which passes it back to the Web browser, which displays the response.

The problem with old solutions

Related topics

While this approach certainly works, it's not the easiest to implement, for a number of reasons.

First is the choice of programming languages. Perl and C fall in the easier-to-write-than-to-read category—compact, powerful, and potentially difficult to maintain. Although these characteristics have a certain charm, the languages don't lend themselves to casual use.

A more important point is that these languages do not have built-in database capabilities. While add-on libraries are available, they simply cannot be as well integrated as products that integrally support databases.

Finally, and most important, the simple post-and-respond paradigm is not suited for a live database connection. Here's why: Once the Web server sends out the form to be filled in, it forgets who you are. One request is treated like any other. In other words, it is stateless. Also, each CGI process is started, executes, and terminates. Therefore, the Web server and CGI program cannot assume anything about the response. There must be enough information in the form to process the request. For example, you can do a simple posting, because that's a new row of data not connected to anything; and you can do a query, because all you need are the query parameters. But you cannot browse through a table, going from one row to the next, because neither the Web server nor the CGI program remembers where you left off.

There are solutions to these problems. You could add state management to your Perl and C programs with yet another add-on library. Once this library is mastered, that would solve the stateless problem; however, you'd still be left with the other limitations.

Instead of Perl and C, you can use a programmable database like Visual dBASE or Paradox that have built-in database capabilities. In addition to their native database power, these products are easier to use and master. But they still require code, which could be provided in an add-on library, to handle the CGI, HTML, and state processing, just as Perl and C do.

IntraBuilder and dynamic HTML

Related topics

IntraBuilder solves all of those problems.

- Database connectivity
- Persistence
- State management

Database connectivity

Related topics

IntraBuilder has drag-and-drop database connectivity.

Drag a table from the IntraBuilder Explorer and drop it onto a form to open it. You've just created a live Query object linking the form to the table.

Now drag an active field from the Field palette onto the form, and you have a form component that displays the table data, automatically updates as the user navigates, allows data entry into the field, automatically saves the changes when necessary, and allows for query and search by form; all with a few clicks of the mouse.

IntraBuilder's database capabilities are on par with the most advanced programmable databases on the market.

Persistence

Related topics

The IntraBuilder Server runs as a continuous process, in parallel with your Web server. Unlike a separate CGI program that must be started by the operating system every time a request comes in, once started, the IntraBuilder Server alertly waits for something to do. This reduces overhead, increases performance, and makes state management a snap.

State management

Related topics

Running as a persistent process, the IntraBuilder Server automatically maintains the state of each user accessing the Web application.

The magic of just-in-time HTML

Related topics

To get a sense of how database applications work over the Web, it's helpful to understand the scope of each component. For a live database Web application, you need:

- A Web browser, running on the client's machine
- A Web server, running the Web site
- The IntraBuilder Server software

The following events occur when a user on the Web loads an IntraBuilder form:

1. The Web browser requests an IntraBuilder form or report as a result of the user typing the URL directly into the browser or clicking a link on an existing Web page.
2. The Web server, which has been configured to recognize an IntraBuilder request, passes that request onto the IntraBuilder Server.
3. The IntraBuilder Server runs the JavaScript code that defines the form. The contents of the form, which were designed as a 2-dimensional coordinate-based form, are rendered into a table-based HTML form.
4. This dynamically formulated HTML is sent back to the Web server.
5. The Web server sends the output to the requesting browser, which displays the form to the user.

The IntraBuilder Server maintains the form internally after it has been created. IntraBuilder's copy of the form contains live links to tables, just as a LAN-based data-entry application would. After the HTML version has been sent to the Web server, the IntraBuilder Server waits for the next request.

Meanwhile, the Web browser is displaying the form to the user. At this point, users are free to treat this form like any other HTML page. They can do nothing and let it sit there in their browser, or switch to another application, or simply leave their computer alone. Users can ignore it and go to another page on the Web in their browser. Or they can use the form.

Using the form includes filling in any data-entry components on the form, and clicking a button or image to submit the form. For example, users can edit fields and click a button to save the changes and display the next row.

Until the user submits the form, there is no communication between the form on the browser and the IntraBuilder Server. The IntraBuilder Server does not know if the user discarded the form, if and when they started editing, what field the user is in, or what items they have changed.

This lack of communication might be important, especially if you've developed LAN-based applications. For example, you cannot have the user type in a customer number and then lookup and fill in the customer's name and address when they tab out of the customer number field, because simply tabbing out of a field does not submit the form. But you could have the user type in the customer number and press a button to submit the form, have IntraBuilder Server do the lookup and fill in the customer info, and then return the updated form for further data entry.

HTML forms can be submitted only by clicking a button or image on the form (or simulating that action through client-side JavaScript). Submitting an IntraBuilder form is slightly different than loading a new one through a URL, and follows these steps:

1. The Web browser posts the form to the Web server.
2. The Web server passes on the values in the form to the IntraBuilder Server.
3. Using information that was automatically stored in the form when it was previously sent out, the IntraBuilder Server identifies the internal version of the form that matches the one that was sent in from the Web browser.
4. The contents from the Web browser update the internal copy of the form, as if those values were typed in a traditional data-entry program.
5. The programmed action of the clicked button or image now occurs, as if that button or image were clicked on the IntraBuilder Server. This action can do anything, like move to the next row in the table

or start a search.

6. The contents of the form, which may have been changed by the action, are again rendered into HTML and sent to the Web server.
7. The Web server passes the updated form back to the Web browser, which displays the form to the user.

Like any Web transaction, all of this can be instantaneous or take several seconds, depending on many factors such as the speed of the server and the load on the Web.

Choosing your development directory

Related topics

Before you can begin developing your IntraBuilder application, you must decide where to put it. When the application is done and ready for real-world use, its files must be deployed in a place that the IntraBuilder Server can access them. There are two approaches:

1. Develop the application in the deployment directory. The advantage with this approach is that the files don't have to be moved. It facilitates testing during development, and once the application is done, it does not have to be retested after being deployed, because it hasn't been moved.

   This approach is the more common one, especially when using the same machine for both development and deployment. With the IntraBuilder Server and Web server active, changes can be made in the IntraBuilder Designer, then you can switch to a Web browser to test the changes immediately, all on the same machine.

   A variation of this approach is to map the directory on the deployment, or server, machine to the development machine. Then you have the IntraBuilder Server and Web server running on the server machine, and the IntraBuilder Designer running on the development machine, but with changes being made directly to the files on the server.

   The disadvantage to this approach is that the development process can create many unwanted files —backups, discarded attempts, etc.—that would clutter up the deployment machine. In addition, test data must be removed before the application is made available for public use.

   In some situations, developers are not allowed to have direct access to Web server machines, so direct development on the deployment machine is not an option. In that case, the second approach is used.

2. Develop on one machine, then copy the files to the deployment machine. You can still do testing on the development machine, using the single-machine approach described before, but you should also test the application after it has been deployed to the server machine.

   The files that need to be copied are detailed in the deployment section of the IntraBuilder Server Help file. Any path names that are different on the two machines must be updated. You can simplify matters by using the same path names on both machines. For example, the IntraBuilder Server can be installed in the default directory on the C drive of the server machine, and the IntraBuilder Designer can be installed in the default directory on the C drive of the development machine. As long as the necessary files are copied, the same path names can be used during development and deployment, although the files are actually on different machines.

Designing tables introduction

Related topics

You may already have many tables, and perhaps different database systems, both local and remote, that you will connect to IntraBuilder forms deployed on your intranets. Yet in some cases it is convenient to quickly create a new table within IntraBuilder (using either the Table Expert or Table Designer). You can also use the Table Designer to modify an existing table before deploying it.

This section of the Help file explains how to use the Table Designer. Guidance is provided in

- Restructuring tables
- Setting data-entry constraints and referential integrity on tables that support it (such as Paradox)
- Creating sorts and indexes, including specific instructions for indexing DBF and DB tables

Note The terms “database” and “table” are often confused. A database consists of one or more tables that may be related by key fields.

Using the Table Expert

Related topics

To use the Table Expert to create a new table,

1. Choose File|New|Table (or double-click the (Untitled) icon on the Tables page of the IntraBuilder Explorer). The New Table dialog box appears.
2. Choose Expert.
3. Select the fields you want from the available sample tables, and your new table is created for you.

Click the Help button on any step of the expert for help with that step.

Using the Table Designer

Related topics

To create a new table using the Table Designer,

1 Do one of the following:

- Choose File|New|Table.
- Drag the (Untitled) table icon from the Tables page of the IntraBuilder Explorer and drop it onto the desktop.
- Double-right-click the (Untitled) table icon on the Tables page of the IntraBuilder Explorer.
- Select the (Untitled) table icon on the Tables page of the IntraBuilder Explorer and then click the Designer button in the tool bar.

Any of these actions displays the New Table dialog box.

2 Click the Designer button. The Table Designer appears.

A default template for the first field is displayed with the Name column highlighted.

3 Set the table type. (See Table structure concepts **for an explanation of the differences between the standard Paradox and dBASE table types.)**

4 Type a name (no spaces for dBASE files) in the highlighted Name field. (You have to name a field before specifying any of its other characteristics.)

5 Specify values for the remaining attributes (Type, Width, Decimal, Index) by typing what you want, or by selecting a value from the drop-down list, or by clicking the spinbox arrows.

6 Create additional fields by pressing the down arrow key (or right-click and choose Add Field from the menu).

You can generate new fields in rapid succession by naming each, then pressing the down arrow key. After naming all the fields in your table design, you can go back and set or reset the attribute values for each field.

Tips

- To add, insert, or delete fields, right-click in the Table Designer window to display a menu, and choose the appropriate command. See Adding and inserting fields.
- To reorder the sequence of fields, place the insertion point in the field number box—it becomes a hand—and move the field to the desired position in the list.
- If you need more information on elements of the Table Designer, see The Table Designer window.
- If you need information on DB and DBF table types and field attributes, see Table structure concepts.

The Table Designer window

Related topics

This section defines the elements of the Table Designer in detail.

To open the Table Designer to create a new table, see Using the Table Designer.

To open the Table Designer to modify an existing table, double-right-click the table on the Tables page of the IntraBuilder Explorer. Or, select the table name on the Tables page and either right-click and choose Design Table from the menu, or click the Designer button in the toolbar.

The top portion of the Table Designer represents the properties that apply to the entire table:

- **The title bar** shows the name of the table, or Untitled if you've not yet saved a new table design. You'll give the table a name when you save it.
- **Updated** is the date the table was last updated and saved.
- **Rows** is the number of rows (records) in the table.
- **Bytes Used** is the total number of bytes used by the fields defined in the table.
- **Bytes Left** is the maximum record size for the selected table type, minus the bytes used for fields. (One byte is used to mark deleted records).
- **Type** is the type of table. You can always create Paradox (DB) and dBASE (DBF) tables. These are considered Standard table types by the Borland Database Engine (BDE). If you select Paradox, a Field Properties Inspector appears on the IntraBuilder desktop.

  Other database types (such as Access or various SQL databases) may be available if you have configured them at the server with the BDE Configuration Utility. (Check the level of client/server support offered by your edition of IntraBuilder.)

The lower portion of the Table Designer lists the fields defined in the table. Each field in the table appears in a separate row.

You define the attributes for each field:

- **Field** contains a number that identifies the field in the table. Field numbers are consecutive, automatic, and read-only. They determine the default order in which fields appear in the Table Records window.
- **Name** is the name of the field (up to 10 characters for DBF fields; up to 25 for DB). You can enter letters, numbers, and underscores, but no other characters. The first character must be a letter. DB and most SQL tables allow spaces; DBF does not.
- **Type** is the field type. Select the type you want from the list. The type you select determines what kind of data the field will contain. It also determines whether you can set the width, decimals, and index options for this field.
- **Width** is the field size. In the case of DBF tables you can change field size for character, numeric, and float fields only (all others have fixed width).
- **Decimal** is the number of digits allowed to the right of the decimal point (for float and numeric fields only). In the case of DBF tables, float and numeric fields have no decimals selected, by default. You can set decimals to a maximum of 2 less than the width value you define. The total width must be 20 characters or less. This includes decimal settings, the decimal point, and an optional minus sign.
- **Index** determines whether to index records using the values in this field (you can set an index on character, date, float, and numeric fields in DBF tables). Select Ascend to index this field in ascending order (for character fields, this is ASCII order, or the order determined by your language driver). Selecting Descend indexes this field in descending order, and None (the default) omits this field from indexing (or removes an existing index associated with this field).

  If you select Ascend or Descend for a dBASE table, the Table Designer creates an index for the field in the multiple index file (.MDX) associated with the table.

  To set a primary key on a Paradox table, choose Structure|Define Primary Key.

For information on setting up SQL Links for those editions of IntraBuilder that support it (Professional and Client/Server), see Connecting your IntraBuilder application to SQL servers.

Getting around in the Table Designer

Related topics

In the Table Designer, each row represents one field (or column) in the table you are designing or modifying. To add, change, or delete data, first select the field.

Using the mouse, click the field you want to change. Using the keyboard, use the following keys:

Getting around in the Table Designer

To go to	Press these keys
Next column	Tab or Enter
Previous column	Shift+Tab
First row	Ctrl+PgUp
Last row	Ctrl+PgDn
Next row	Down arrow
Previous row	Up arrow

The field appears highlighted as you select it.

To go to a specific field number, choose Structure|Go To Field Number or press Ctrl+G. Type the number of the field to go to, and choose OK.

Adding and inserting fields

Related topics

You can add new fields to the table by either adding a row at the end of the fields list or by inserting a row anywhere in the list.

To add a new field to the end of the fields list, choose Structure|Add Field (or right-click anywhere in the Table Designer and choose Add Field from the menu).

To insert a new field between other fields, select a row, and choose Structure|Insert Field, or right-click and choose Insert Field from the menu. The new row appears above the row you selected.

Moving fields

Related topics

To move a field, changing its order in a table, point to the field number in the leftmost column. When the pointer changes to a hand, drag the row up or down to its new location.

Deleting fields

Related topics

To delete fields from a table,

1. Click anywhere in the row of the field you want to delete.
2. Choose Structure|Delete Current Field (or right-click and choose Delete Current Field from the menu).

The Table Designer deletes the field definition. If the table contains records, the data in this field is deleted as soon as you save the table structure.

Viewing a table’s properties

Related topics

To view a table’s properties,

1. In the IntraBuilder Explorer, on the Tables page, select the table name.
2. Click the right mouse button to display a context menu.
3. Choose Properties.

The File Item Properties dialog box appears.

You can view the properties of other IntraBuilder files, including forms, reports, and queries, the same way.

Adjusting the Table Designer window

Related topics

You can resize or move columns, move rows, and hide grid lines in the Table Designer.

- To resize a column, point to the column border. When the pointer changes to a double-headed arrow, drag the border until the column is the size you want.
- To move columns, point to the title of the column you want to move. When the pointer changes to a hand, drag the column to its new location.
- To show or hide grid lines, choose Properties|Table Designer Properties, or right-click and choose Table Designer Properties from the context menu. The Table Designer Properties dialog box appears. Check or uncheck the Horizontal Grid Lines or Vertical Grid Lines, as you want.

Saving the table structure

Related topics

Save the table design to keep the structure you've created. If you haven't yet saved a new table design, doing so creates the table and any associated files (such as DBT and MDX files).

To save changes to a table design, do one of the following:

- If it's a new table, choose File|Save.
- To save an existing table under a new name, Choose File|Save As.

If you are saving for the first time, or chose Save As, the Save Table dialog box appears.

Type a valid file name. Choose a destination drive and directory, if needed, and then choose OK. IntraBuilder creates or updates the table and any associated files.

Note You cannot use a file-name extension ending in a 'T'.

Abandoning changes

Related topics

Abandon changes to a table design if you want to cancel creating a new table or discard the changes you have made to an existing table.

To abandon changes,

1. Choose File|Close or press Esc to close the Table Designer.
2. Choose No when asked to save changes.

Printing the table structure

Related topics

When you've finished designing a table, you might want to print the table structure for future reference. To do so, open the table in the Table Designer, choose File|Print, choose the print options you want, then choose OK.

Restructuring tables

Related topics

It’s easy to change the structure of a table—even if the table contains records.

If the table is empty, you can make any valid changes you want to the table structure except change the table type. If the table contains records, however, you need to be more careful about the changes you make—and you should make a backup copy of the table before attempting to change its structure.

When you change the structure of a table, the Table Designer makes a backup copy of the old table, creates a new table with the revised design, and attempts to copy all the data from the backup table to the new table. (Each time you change the structure of this table, the backup copy that the Table Designer created is overwritten. That is why you should make your own backup copy with a unique name or in another directory.)

Important guidelines for restructuring

Related topics

When you change the structure of a table, the Table Designer uses the field name and field position to determine how to transfer information to the new structure.

Warning! If it cannot find a corresponding field in the new table, the Table Designer *does not copy the data from the fields in the backup table*; instead, the information is lost when the backup table is deleted.

To prevent losing data that you want to keep, save the table structure frequently as you make changes and confirm that they are completed successfully.

If you change the type of a field, the Table Designer does its best to convert data to the new type. Some conversions are relatively straightforward, such as converting date, logical or numeric fields to character. However, radical conversions (such as a memo field to a date field) might produce results you don't want. In addition, the Table Designer does not copy data that is invalid in the new field type. For example, attempting to copy the value "123ABC" from a character field to a numeric field fails because letters aren't valid entries in numeric fields.

In addition to these guidelines, remember that if you delete a field in a table that contains records, you lose the information in that field permanently. You can recover the information only if you have made a backup of the table.

Changing the structure

Related topics

To change the structure of a table,

1. Open a table in Design mode. To do this, choose File|Open, click the table name, and select Design Table Structure. Or, if the table is already open in a Table Records window, choose View|Table Design.

   The Table Designer opens, displaying the table's current definition. (If you are working in a shared environment, you see a prompt to open the table exclusively. Choose Open Exclusive to open the Table Designer.)
2. Make a copy of the table to work on (choose File|Save As and specify a new name for the table).
3. Change the field definitions you want. You cannot change the table type.
4. When you finish, choose File|Save. In addition to saving your changes, the Table Designer also copies associated files (such as MDX and DBT files).

Note Open the restructured table in the Table window to verify that your data is in the condition you want. If not, you can revert to your original table if you worked from a copy.

Table access passwords

Related topics

In addition to restricting access to intranet sites, you can limit access to sensitive tables by setting passwords directly on those files. IntraBuilder generates automatic password forms that prevent unauthorized access to encrypted tables.

For more on IntraBuilder security features, see Setting up security.

Specifying data-entry constraints

Related topics

If supported by the database type of your database server software, you may be able to specify data-entry constraints—rules that govern the values you can enter in a field. If you want to make sure that the values users enter in a field meet certain conditions, specify a data-entry constraint for that field.

You can specify data-entry constraints in the Inspector when you create or modify a table that supports them, such as a Paradox table (data entry constraints are called validity checks in Paradox). DBF table do not support data-entry constraints.

The Inspector displays different data entry constraints depending on the field type.

Data-entry constraints

Validity check	Meaning
Required	Every record in the table must have a value in this field.
Minimum	The values entered in this field must be equal to or greater than the minimum you specify here.
Maximum	The values entered in this field must be less than or equal to the maximum you specify here.
Default	The value you specify here is automatically entered in this field. You can replace it with another value.

Referential integrity

Related topics

Referential integrity means that a field or group of fields in one table (the “child” table) refers to the key of another table (the “parent” table). Only values that exist in the parent table’s key are valid values for the specified field(s) of the child table.

You can establish referential integrity only between like fields that contain matching values. For example, you can establish referential integrity between the sample CUSTOMER.DB and ORDERS.DB tables on their Customer No fields. The field names do not matter as long as the field types and sizes are identical.

IntraBuilder lets you establish referential integrity for any file type that supports it. You cannot establish referential integrity between DBF tables; however, you can use DB tables if you need referential integrity. You can also use some SQL server tables if you need referential integrity. See your server documentation to determine if your table type supports referential integrity.

Defining referential integrity

Related topics

You can establish referential integrity between tables in the current database. If no database is specified, you can establish referential integrity between tables in the current directory.

To define a referential integrity relationship,

1. In IntraBuilder Explorer, Tables page, use the Look In box to select a current database login or a directory containing tables (such as DB type) that support referential integrity.
2. Choose File|Database Administration. The Database Administration dialog box appears
3. Specify a Table Type that supports referential integrity, such as Paradox, then click Referential Integrity. **The Referential Integrity Rules dialog box appears**
4. Choose New.

   The New Referential Integrity Rule dialog box appears. All tables in the current database or directory appear in the Parent Table and Child Table drop-down lists

   Choose a parent table from the Parent Table list. The table's key fields appear in the Primary Key Fields area of the dialog box.
5. Choose the child table from the Child Table list. Fields available for referential integrity appear in the Available Child Fields list.
6. Specify whether the tables are in a one-to-one or one-to-many relationship in the Relationship panel. The relationship you choose changes the available child fields.

- One-to-one relationships can be defined between the primary key field in the parent and the primary key field in the child, or any field in the child that has a unique index.
- One-to-many relationships can be defined between an indexed field that is not the primary key in the child and the primary key field in the parent.

7. Choose the child table's field in the Available Child Fields list and click the Add Field arrow. The field name appears in the Related Child Fields area of the References panel.

   You can establish referential integrity with a composite key. If the parent table has a composite key, add fields from the Fields list to match all of the fields in the parent's key.
8. Select the update and delete behavior you want. (See Update and delete behavior**.)**
9. Optionally change the rule name IntraBuilder provides in the topmost box.
10. Choose OK to save the referential integrity relationship.

Note If you attempt to define referential integrity on a table that already contains data, some existing values may not match a value in the parent's key field. When this happens, the operation fails to complete and you receive an error message.

The first rule says that the key field “Customer No” in the child table “orders.db” refers to the same field in the parent table “customer.db”.The second rule that the key field “Order No” in the child table “lineitem.db” refers to the same field in the parent table “orders.db”.

Update and delete behavior

Related topics

You can specify the following rules for updating and deleting data in a parent table that has dependent records in a child table:

- Restrict: You cannot change or delete a value in the parent's key if there are records that match the value in the child table.

  For example, if the value 1356 exists in the Customer No field of Orders, you cannot change that value in the Customer No field of Customer. (You can change it in Customer only if you first delete or change all records in Orders that contain it). If, however, the value doesn't exist in any records of the child table, you can change the parent table.

- Cascade: Any change you make to the value in the key of the parent table is automatically made in the child table. If you delete a value in the key of the parent table, dependent records in the child table are also deleted.

The availability of cascading updates and deletes varies according to the table type:

- Paradox: Cascading updates only
- Oracle: Cascading deletes only
- Sybase: No cascading updates or deletes permitted
- InterBase: No cascading updates or deletes permitted
- Microsoft SQL Server: No cascading updates or deletes permitted

Changing or deleting referential integrity

Related topics

You can choose any referential integrity name from the list of named referential integrity relationships in the Referential Integrity Rules dialog box to either modify or delete it.

- Choose Edit to open the Edit Referential Integrity Rule dialog box with the selected referential integrity relationship filled in. You must be able to obtain exclusive access to all tables involved in the referential integrity when you
modify it.
- Choose Drop to delete the selected referential integrity relationship.

Table structure concepts

Related topics

Before you actually create each table, think through the table structure on paper first. When you create a table, you define its structure, which includes the table name, table type, and the names and attributes of individual fields.

Table names

Related topics

See your database software documentation to determine valid file names for its tables. For example, an Access table has no OS-enforced extension requirement because it is stored within an Access database with an MDB extension. On the other hand, DB is the required extension for Paradox tables and DBF for dBASE tables.

The table name should indicate its purpose and be easy to remember. For example, if a table contains employee information, you might call it EMPLOYEE.DBF or STAFF.DBF.

Table types

Related topics

The *table type* determines the file format of a table.

The table type you define depends on the way you plan to use the table and the types of database management software your installation will support. If you expect to use the table only with IntraBuilder applications, then choose either the DB or DBF format. Use DB to take advantage of referential integrity. Use DBF if you want to work with expression indexes.

However, if the table is to be shared with other applications, consider the most useful format for all applications involved.

All versions of IntraBuilder support local Paradox tables, dBASE tables, and any table that you can access through ODBC, such as a Microsoft Access database. The IntraBuilder Professional Edition includes BDE and SQL Links for Borland InterBase and Microsoft SQL Server. If you have the Client/Server edition of IntraBuilder (which includes BDE and the complete SQL Links) you can also work with Sybase, Oracle, Informix, Microsoft SQL Server, IBM DB/2, and Borland InterBase databases, as well as remote ODBC.

The IntraBuilder interface adjusts automatically to accommodate the type of table you are using. For example, if the table with which you are working supports it, you can specify data entry constraints in the Inspector while working in the Table Designer. Otherwise, data entry constraints are unavailable in the Table Designer.

Field types

Related topics

Each field has a defined *field type*, which determines the kind of information it can store. For example, a character field accepts all printable characters including spaces. You can define up to 1,024 fields in a table.

An IntraBuilder DBF table can contain the following field types.

DBF field type

Field type	Default size	Maximum size	Index allowed?	Allowable values
Character	10 characters	254 characters	Yes	All keyboard characters
Numeric	10 digits, 0 decimal	20 digits	Yes	Positive or negative numbers
Float	10 digits, 0 decimal	20 digits	Yes	Positive or negative numbers
Date	8 characters	N/A	Yes	Dates in a valid date format, such as MM/DD/YY
Logical	1 character	N/A	No	True (T, t), false (F, f), yes (Y, y), and no (N, n)
Memo	10 characters	N/A	No	Usually just text, but all keyboard characters; can contain binary data (but using binary field is preferred)
Binary	10 characters	N/A	No	Binary files (sound and image data, for example)
OLE	10 characters	N/A	No	OLE objects from other Windows applications

The field type determines what you can do with the information in the field. For example, you can perform mathematical calculations on values in a numeric field, but not on values in a logical field.

The field type also determines how the data appears in the field. For example, a date field, by default, displays dates in the MM/DD/YY format (such as 02/14/96). The display of field data is also affected by the settings of the Windows Control Panel and the settings defined by using the BDE Configuration Utility.

DB tables have different field types with different rules, as described in Figure 3.3.

DB field types

Field type	Field size	Description
Alpha	1–255 (required)	Contains letters, numbers, special symbols (like%, &, #, and =), or any other printable character.
Number	8	Contains numbers in the range –10307 to 10308 of up to 15 significant digits.
Money	8	Contains numbers in the range –10307 to 10308 of up to 15 significant digits
Short	2	Contains whole numbers in the range –32,767 to +32,767
Long	4	Is a 32-bit signed integer. Contains whole numbers (nonfractional) with complete accuracy in the range –2147483648 to +2147483647 (plus or minus 2 to the 31st power).
BCD	0–32 (number of digits after the decimal point)	Contains numeric data in a BCD (binary coded decimal) format. The BCD field type is provided primarily for compatibility with other applications that use BCD data.
Date	4	Contains any valid date (including BC dates) to December 31, 9999
Time	4	Contains times of day, stored in milliseconds, since midnight, and limited to 24 hours. This field type is read-only in IntraBuilder.
Timestamp	8	Contains both a date and time value.

Memo	1–240 in DB file; unlimited in MB file	Contains free-form, variable length text
Formatted memo	1–240 in DB file; unlimited in MB file	Like memo fields, except that they also contain text formatting (font, styles, colors, sizes, tabs, justification, and so on)
Graphic	1–240 in DB file; unlimited in MB file	Contain graphic images created using another application
OLE	1–240 in DB file; unlimited in MB file	Contain OLE objects from another Windows application
Logical	1	Contains values representing *true* or *false* (yes or no)
Autoincrement	4	Contains long integer values in a read-only (non-editable) field, beginning with the number 1 and automatically incrementing. Deleting a record does not change the field values of other records.
Binary	1–240 in DB file; unlimited in MB file	Contains data that IntraBuilder can't interpret or display. A common use of a binary field is to store sound.
Bytes	1–255	Contains data that IntraBuilder can't read or interpret. A common use of a bytes field is to store bar codes or magnetic strips. Unlike binary fields, bytes fields are stored in the DB file (rather than in the MB file), allowing for faster access. IntraBuilder does not read or write this field type.

Other table types, such as SQL server tables, may have different field types. Refer to your SQL-based server documentation for specific details.

Indexing and sorting IntraBuilder tables

Related topics

Records in local IntraBuilder DBF and DB tables can be organized either by indexing or by sorting. Both methods arrange records in a specific order, but in completely different ways. This section describes the methods of organizing rows (records) in a local IntraBuilder table. It covers the following topics:

- Indexing vs. sorting
- Simple indexes and complex indexes
- Design concepts and guidelines for indexes
- Adding, modifying, and deleting indexes
- Sorting data to a separate table
- Creating indexes for DB tables

Note The material in this section applies to DBF, DB, and SQL indexes. However, specific guidelines and procedures might differ. If you're using SQL tables, see your SQL documentation.

Indexing versus sorting

Related topics

Indexing and sorting are two approaches for establishing the order of data in a table. You use them to answer different needs in an application. In general, you index a table to establish a specific order of the rows, to help you locate and process information quickly. Sort only when you want to create another table with a different natural order of rows.

Indexing orders rows in a specific sequence, usually in ascending or descending order on one field. Indexing creates a list of rows arranged in a logical order, such as by date or by name, and stores this list in a separate file called an *index file*. An index (MDX) file can have up to 47 indexes, but only one controls the order of rows at any time. The index that is controlling the order is the current master index.

Sorting creates an entirely separate copy of the current table with the rows in a different order. You're likely to use sorting infrequently, only when you want to create a separate table with a different natural order.

Multiple index files. IntraBuilder stores indexes in multiple (MDX) index files, and recognizes older MDX files. You can design and maintain multiple indexes using the Manage Indexes dialog box.

Here is a summary of key differences between indexing and sorting:

- **Creating tables.** Indexing creates an index file that consists of a list of rows in a logical record order, along with their corresponding physical position in the table. Sorting a table creates a separate table and fills it with data from the original table, in sorted order.
- **Arranging rows.** Both indexing and sorting arrange rows in a specified order. However, indexing changes only the logical order and leaves the natural order intact, while sorting changes the natural order of the rows in the new table.
- **Processing operations.** Certain operations are much faster using indexes, such as searching for data, running queries, and so on. Some operations, such as linking tables, require indexes.
- **Using functions.** With indexes, you can order rows using fields and JavaScript methods. With sorting, you can use fields only, in ascending or descending order.
- **Adding rows.** If you add rows to an indexed table, the index is updated automatically so that the rows appear in the correct order. If you add or change rows in an already-sorted table, you might need to sort it again.
- **Mixing field types.** With indexing, you must convert field values to a common field type, for example, converting the sale date to a character type. With sorting, you can order rows on fields with different field types; for example, you can sort on customer number (a character field) and sale date (a date field), without converting them to a common field type.
- **Mixing orde**r. With indexes, the entire index is either ascending or descending. With sorting, you can mix fields sorted in ascending and descending order.

In general, use indexing to make processing more efficient in data entry, forms, queries, and reports. The only significant costs are that index files require extra disk space, and processing time is required for ongoing automatic maintenance.

Sorting rows

Related topics

Sorting a table copies its contents to a separate table and arranges rows in the order you specify in the new table. When you sort, the *source table* is the table containing the rows you want to copy, and the *target table* is the new table to contain the copied rows. Sorting does not change the data in the source file.

When you sort a table, all fields in the source table appear in the target table. You select the fields on which to sort records. You can also select the records you want to include in the target table by using the scope options.

IntraBuilder sorts data in case-sensitive alphabetic order, using the sort order specified by the language driver in the BDE Configuration Utility. Sorting starts with the first character in the key and proceeds from left to right. Punctuation comes before numbers, numbers before letters, and uppercase letters before lowercase letters.

Tip In general, use sorting only when you want to export data to another application or to create a separate table for reporting or other purposes. Use indexing instead when you want to make data entry, querying, and reporting tasks faster and more efficient.

Note Make sure you have enough available disk space to store the table on the target drive.

To sort data,

1. Open the table you want to sort in Run mode.
2. Choose Table|Sort Rows. The Sort Rows dialog box appears.

Figure 3.1 Sort Rows dialog box

3. Select the field(s) on which to sort records, and click the > button to move them to the Key Fields list.

   The order in which the selected fields appear in the Key Fields list determines the order of the sort. In the example, records would be sorted first by zip code, then by name. The target table contains all fields from the source table.
4. Select each key field, then specify the sort order and whether the sort is case sensitive.

   The Scope options let you select the records you want to include in the target table.
5. When you have finished, click OK. IntraBuilder creates a new table. If the target file exists, IntraBuilder asks whether to overwrite it. The records you selected are copied to the target table and sorted as you specified, starting with the first key field.

Planning indexes

Related topics

When you design indexes for a table, consider how you will use and process data. Indexes affect and support features that an application provides: data entry, queries, and reports. Asking the right questions at the beginning can save you redesign efforts later.

Using indexes in data entry

Related topics

Because indexes affect the order in which records appear, they let users find and update information quickly. To make data entry more efficient, consider these questions:

- What is the order in which users expect to see the data? For example, they might expect to see a list of companies in alphabetical order, a list of purchase orders by purchase order number, or a list of invoices in chronological order. Indexes should reflect the *expected* order of information in a table. If users expect the same information in different sequences, you can create multiple indexes—one for each sequence. For example, in the Orders table, you might want separate indexes for the order number, order date, and customer number.
- To find records in a table, what kind of information might users know already? For example, to locate an invoice, users might already have the invoice number, approximate date of the invoice, or the company that submitted the invoice. To speed up the search process, you might want to create indexes for the most common ways a user looks for information.
- What kinds of calculations are users going to perform on data in the table? For example, users might want to calculate the average sale per state or the total sales per month. The word “per” is a clue to an index you might want to create—in the first example, indexing the state field and, in the second example, indexing the sales date field. An index can put similar records in consecutive order so that users can quickly search for the first record in the series and stop processing after the last record in the series. For example, if users want to calculate the total payments to a vendor, consider creating an index for the vendor number or name.

Using indexes in queries

Related topics

Indexes can increase the speed at which a query is processed. Indexes are also required for defining links among related tables. To make queries more efficient, consider the following issues:

- What kinds of questions are users going to ask? For example, will they want to know the number of items in stock for a particular product? If so, consider creating an index for the product name or identification number.
- What kind of information might a user know before attempting the query? For example, a user might know the name of the product, its identification number, or its type. Consider creating indexes for commonly known information.
- If the index is solely for occasional or ad hoc queries, consider generating an index at query time instead of maintaining an index separately on an ongoing basis. When the query is finished, you can delete the index to recover disk space.

Using indexes in reports

Related topics

Indexes affect the order in which records appear in a report. In addition, they can trigger subtotals and totals in a report (when key values change). To make reports easy to design, consider the following issues:

- What is the order in which users expect to see information in the report? For example, do users want to see a chronological list of invoices billed? An index can ensure that records appear in the expected order.
- What kinds of calculations will the report make? For example, a report might show the total number of sales by salesperson, or the average sale by customer. The word “by” is a clue to an index you might want to create—in the first example, indexing on the salesperson field and, in the second example, indexing on the customer number. Using an index makes it easier to calculate running totals. If a report includes subtotals within totals, consider using a complex index.
- If the index is solely for occasional or ad hoc reports, consider generating an index at report time instead of maintaining an index on an ongoing basis. When the report is finished, you can delete the index to recover disk space.

Using indexes to link multiple tables

Related topics

Indexes are required for linking related tables together in a multi-table query. To link tables, consider the following issues:

- What are the relationships among the tables—one-to-one, one-to-many, many-to-many? For example, in the sample tables, the Orders table and the LineItem table are in a one-to-many relationship. The Orders table is the parent table and the LineItem table is the child table.
- With related tables, which fields are common among them? To link tables together, you must have an index for the child table on a field that also appears in the parent table. For example, the Orders table and LineItem table both have an ORDER_NO field, and the LineItem table has an index on this field.
- Can you use codes instead of long character fields? For example, to link orders in the Orders table to customers in the Customer table, the application uses the customer number, a short character field that uniquely identifies each customer.

DBF index concepts

Related topics

Before you create indexes on DBF tables, you need to be familiar with a few general concepts.

- Multiple index (MDX) files**.** When you create an index, it is stored in a file with the file-name extension MDX. Each index has a name (sometimes called a *tag*) that defines the index uniquely in the MDX file.

  A table's main MDX file is called the *production index* file. The production index file opens automatically when you open a table, so its indexes are automatically available—though no index sets the record order until you select it as the master index. As you update records in a table, the affected indexes in the production index file are also updated. If you use any non-production MDX files, they must be opened explicitly by entering statements in the Script Pad.

  The production index file has the same name as the table plus the MDX extension.

- Key expressions. A key expression is a field name, or a combination of field names, functions, or operators, that determines how an index orders records in a table. It must be a character, numeric, date, or float field, or an expression that evaluates to one of these types. The key expression can be up to 220 characters in length.
- Simple indexes. A simple index uses a single field name for the key expression.
- Complex indexes. A complex index uses a combination of one or more fields, or a DBF expression.
- Ascending and descending order. Records can be ordered in ascending order, lowest to highest (the default), or descending order, highest to lowest. For character fields, the order is ASCII or the order established by the language driver installed by the BDE.

Note Keeping a large number of indexes affects performance, because IntraBuilder must update each one as the table is revised. If you need to improve performance, consider removing rarely used indexes from the production index file.

Creating a simple index

Related topics

A simple index consists of a single field.

The key of a simple index is just the name of a field. For example, in the Customer table, if you index on the CUSTOMER_N field, the key is the field name, CUSTOMER_N.

You can create a simple index using either the Table Designer or the Manage Indexes dialog box, as shown in the next two sections.

Using the Table Designer to create a simple index

Related topics

To create a simple index in the Table Designer, choose an index order for the field you want to use—ascending or descending.

Using the Manage Indexes dialog box to create a simple index

Related topics

To open the Manage Indexes dialog box, in the Table design mode, choose Structure|Manage Indexes. The Manage Indexes dialog box appears.

To create a simple index,

1. Choose New. The Define Index dialog box appears.
2. Choose fields from the Available Fields list and add them to the Fields Of Index Key list at the right.
3. Choose Ascending or Descending order.
4. Choose Specify from Field List for a simple index.
5. Enter a name for the new index.

You can use letters, numbers, and underscores, but the first character must be a letter. The name you use must be unique within the index file. For a simple index, use the field name.

Check your vendor documentation for other limitations. DBF and DB file types are described in Table structure concepts.

By default, IntraBuilder indexes records in ascending order. The exact sort order depends on the driver specified in BDE.

When you choose OK in the Manage Indexes dialog box, IntraBuilder builds any indexes you created or changed and removes any indexes you deleted.

Note You might have to wait while the indexes are created, particularly if the table has many records or if key expressions are long and complex.

To select an index for a table

Related topics

When you first open a DBF table, it appears in natural order.

When you first open a DB table, the natural order is the primary key order.

For DBF tables, the production MDX file opens automatically with the table, but the indexes it contains are not in effect until you select one. To order records in a specific way, select the index you want.

1. Open the form in Form Designer.
2. Select the active Query object
3. Open Inspector on the Query object
4. Select the *rowset* property.
5. Click the *rowset* property tool button.The Inspector displays the rowset's object properties.
6. Set the *indexName* property to one of the available indexes.

Modifying indexes

Related topics

You can modify an existing index to make it more useful or efficient. For example, if you create a simple index for a DBF table in the Table Designer, you might want to make it a complex index by adding fields or expressions. Or, you might learn after using the index for a while that a different key is more suitable.

To modify an index, select it in the Manage Indexes dialog box, and choose Edit. The Define Index dialog box appears. Make your changes, then choose OK.

Deleting indexes

Related topics

You can delete an index you no longer need to save space and improve performance. Deleting an index does not delete any records in the table—it deletes only the separate index that arranges records in a particular order.

To delete a simple index in the Table Designer, do either of the following:

- Choose None as the index type for the field.
- In the Manage Indexes dialog box, select the index you want to remove and click Delete.

The Table Designer removes the associated index from the production index file. If you delete the only index in the file, the MDX file is deleted as well.

Indexing on a subset of records for DBF tables

Related topics

In most cases, indexes include all records in a table. For special circumstances, however, an index might contain only some of the records in a table. Indexing on a subset of records can make it easier to process information in that table. For example, you might want to work with budget information that applies to your sales department only. In this case, you could create an index that includes only those records whose DEPT_ID is SALES.

To create an index that includes only the records you want, first determine which records you want to include, then state this in the form of a valid DBF expression. For example, if you want to create an index of customers in your South sales region only, you could use a For *condition* expression such as SALES_REG = “SOUTH” to create the index. Thereafter, when you use this index, you see and process customers from the South region only.

Hiding duplicate values

Related topics

Indexes can contain multiple records with the same value in an indexed field. For example, the Lineitem table can contain multiple entries with the same ORDER_NO or STOCK_NO.

In certain cases, however, you might want to have a unique index, which finds only the first occurrence of a value in the indexed field and ignores subsequent records with the same value. This kind of index is useful when subsequent records repeat information in the first record.

For example, in the Lineitem table, if all products with the same STOCK_NO were sold at the same price, you could use a unique index to hide duplicate index values, so that only the first record with the price would appear.

If you check Include Unique Key Values Only in the Define Index dialog box, only the first record with a duplicate value in the indexed field is included in the index. Subsequent records with duplicate values in that field are excluded.

Note In DBF indexes, records can have duplicate values in the indexed field. In DB primary indexes, records cannot have duplicate values. In SQL indexes, uniqueness is required if the index is defined as a unique index.

Creating complex indexes for DBF tables

Related topics

Complex indexes on DBF tables use a combination of one or more field names, plus valid DBF expressions. Use a complex index when no single field uniquely identifies each record, or when you need the flexibility of an expression to define the index condition.

Indexes on DB tables also can use multiple fields; such indexes are called *composite indexes*. However, unlike complex indexes in DBF tables, you cannot use functions or operators in the DB index expression.

Rules for DBF complex indexes

Related topics

For complex DBF indexes, the complexity of the index expression varies according to the way the index is used. The following rules apply when defining complex indexes:

- An index value can be up to 100 characters long. The text of the key expression can be up to 220 characters long.
- The complex index must be a valid DBF expression. Note that a single field name is a valid expression.
- The expression must evaluate to a character, date, numeric, or float value.
- It usually, but not always, contains at least one field name.
- For multiple character fields, *concatenate*, or combine, fields using the plus sign (+), as shown in the following examples:

```
LAST_NAME + FIRST_NAME + M_INITIAL
CUSTOMER + ORDER_NO
```

- You can concatenate fields of different data types by converting them to a single type. In the following example, the key expression concatenates the CUSTOMER_N field, which is a character field, and ORDER_DATE, which is a date field. The DTOS() function converts the date value to a character string in the format YYYYMMDD. This order—year first, then month, then day—ensures accurate indexing.

```
CUSTOMER_N + DTOS(ORDER_DATE)
```

- For converting number fields, use the STR() function. Include the width and number of decimal places of the numeric field(s), to ensure accuracy of the index. For example, suppose you are creating an index that includes a character field LNAME, and a numeric field called AMOUNT that is 10 places wide with 2 decimal places. Use the following syntax:

```
LNAME+STR(AMOUNT,10,2)
```

Creating the DBF complex index

Related topics

To create a complex index for a DBF table, choose New in the Manage Indexes dialog box.

You can type a key expression, such as STATE_PROV+CITY, to create a complex index on those two fields. The key expression can use multiple field names, functions, and operators.

The index is saved when you click OK to exit the Define Index dialog box.

Key expressions

Related topics

The following table shows several examples of key expressions and the fields used.

Sample DBF key expressions

Key expression	Fields used	Notes
CUSTOMER_N	CUSTOMER_N	
CUSTOMER_N + ORDER_NO	CUSTOMER_N, ORDER_NO	
CUSTOMER_N + DTOS(SALE_DATE)	CUSTOMER_N, SALE_DATE	DTOS converts date field to character for indexing.
UPPER(LAST_NAME)+UPPER(FIRST_NAME)	LAST_NAME, FIRST_NAME	UPPER changes character field to all caps.

The first example is a single field as the key expression. Complex indexes, on the other hand, can use a combination of one or more fields, plus functions and operators.

- CUSTOMER_N + ORDER_NO is a complex key expression using multiple fields and the concatenation operator (+).
- CUSTOMER_N + DTOS(SALE_DATE) is a complex key expression consisting of multiple field names and a function.
- UPPER(LAST_NAME)+UPPER(FIRST_NAME) converts characters to all caps before concatenating them. The UPPER function prevents sorting problems when capitalized entries are mixed in with lowercase ones.

Primary and secondary indexes for DB tables

Related topics

IntraBuilder lets you create primary and secondary indexes for any table type that supports them. For example, DB tables support primary and secondary indexes.

- A primary index is the main index in a table. It consists of one or more consecutive fields, starting with the first field in the table.
- A secondary index is supplemental to the primary index in a table.

DB and other table types let you specify whether or not a secondary index is case-sensitive. Case sensitivity affects the sort order and the uniqueness of values. In IntraBuilder, you can create case-sensitive indexes only, although IntraBuilder maintains case-insensitive indexes when you edit tables that use them.

DB primary indexes

Related topics

If you are creating DB tables, each table should have one primary index, although it is not required. In a DB table, the primary index is stored in a file with a PX extension.

Unique keys

Related topics

Primary indexes require unique values—they do not permit duplicate key values. For example, if a DB table has a primary index on ORDER_NO, you cannot add two orders with the same order number—only one can exist in the table. In a DB composite index, individual field values can be duplicates, but the combined value of all key fields must be unique. (Secondary indexes do permit duplicate values.)

When you create the primary index, use a field that will contain a unique value for each record, such as the customer number field in the CUSTOMER.DB table.

A table can have only one blank (empty) value in the keyed field, because subsequent blank values are considered duplicates. Therefore, key fields usually require entries.

Field types for key fields

You cannot use the following field types in DB keys: memo, formatted memo, graphic, OLE, binary, logical, or bytes.

Maintained and non-maintained indexes

Related topics

Some tables, such as DB tables, can have two types of secondary indexes:

- *Maintained* secondary indexes are automatically maintained when data changes in the table. IntraBuilder lets you create maintained secondary indexes, and it updates maintained indexes automatically when you edit a table.
- *Non-maintained* secondary indexes are not automatically updated when the table is open. IntraBuilder does not let you create non-maintained secondary indexes, but it supports any existing non-maintained indexes.

You can create maintained secondary indexes only if the table has a primary index. You can create as many single-field (simple) indexes as there are fields in a table, and you can create up to 255 multiple-field (called *composite*) indexes per table.

Creating primary indexes

Related topics

You can create a primary index in the Table Designer or the Manage Indexes dialog box. If the table type you are creating does not support primary indexes (DBF tables, for example), these options are not available. DB tables do support primary indexes.

To create a primary index,

1. Open the table in Table Designer
2. Choose Structure|Define Primary Key to display the Define Primary Key dialog box.
3. Choose the Primary Key fields from the Available Fields list. Click the arrow to add (or remove) fields from the Fields Of Primary Key list box.

Note The first field in the table must be the primary key or part of a composite primary key. If the field you want to be the primary key is not currently the first one in the table, you have to move it up in the Table Designer to be the first field.

Creating secondary indexes

Related topics

If you have created a primary index, you can create one or more secondary indexes in the Manage Indexes dialog box.

1. Choose Structure|Manage Indexes to display the Manage Indexes dialog box.
2. Click the New button. The Define Index dialog box appears.
3. Select fields from the Additional Fields select box and click the arrow to add each one to the Fields Of Index Key box. The double-right arrow adds all the fields at once.
4. Choose Ascending or Descending Order.

Click OK.

Designing forms introduction

Related topics

IntraBuilder forms are deployed by IntraBuilder Server as Web pages that display messages and table records, respond to user input, and provide live access to data by using controls such as data-entry fields, selection lists, check boxes, buttons, and so on. Authorized users can use their Web browser to browse table records, enter new records, and edit existing records. You can build simple data-entry forms based on a single table, or elaborate forms that serve as a complete user interface for complex applications. You can create sophisticated applications for browsers supporting ActiveX, Java, and JavaScript.

To create a form, choose File|New|Form. The New Form dialog box appears, offering two ways to create a form:

- **The Form Expert** saves time by automating the process of adding controls to a form. It presents you with a series of options in dialog boxes, and based on your selections, creates a form, as demonstrated in Quick Tour. Often you might want to begin with the Form Expert to create a basic layout that you can modify and further develop in the Form Designer. See Using the Form Expert.
- **The Form Designer** lets you create forms visually, by selecting functional controls from the Component Palette (such as HTML, data entry fields, list boxes, buttons, and check boxes) and dragging them onto the form. Then you link the controls to the fields of your table and specify properties for the form and its controls. Little or no coding is required. However, you can build quite complex, highly customized applications by setting properties and programming event handlers and methods for each control, or by adding JavaScript code or Java applets to your form.

  While in Form Designer, you can create methods in the Method Editor and you can always test statements and expressions or execute statements in the Script Pad. After you have created a form, you can close the Form Designer and open the Script Editor to directly inspect and edit the complete generated JavaScript code.

This section of the Help file describes the main windows and tools of the Form Designer and offers guidance on using them.

Using the Form Expert

Related topics

To use the Form Expert,

1. Choose File|New|Form. Or, double-click the “full” (Untitled) icon on the Forms page of the IntraBuilder Explorer. The New Form dialog box appears.
2. Choose Expert.
3. Go through the steps of the expert, clicking the Next button when you’re finished with each step. You’ll specify these things:

- The table or query that contains the data you want to use in the form
- The table fields you want to include in the form
- The type of controls you want
- The layout for fields on the form
- The colors and font for the elements on the form

The Form Expert generates the form you specify. At the end of the expert, you have the choice of running the form or opening the form in Design mode to further customize it.

Once generated, you can modify the form as needed, moving or resizing controls, adding graphics, Java or ActiveX applets, creating new controls, and modifying control properties.

Overview of the Form Designer

Related topics

To open the Form Designer from the New Form dialog box (File|New|Form), choose Designer.

To open the Form Designer from the IntraBuilder Explorer, click the Forms tab. (If the IntraBuilder Explorer is not open on the desktop, choose View|IntraBuilder Explorer.)

Then, do any one of the following to open the Form Designer:

- Double-click a form name (or untitled icon).

Important There are two (untitled) icons. The “full” icon opens as a new form with the JFM extension. The “empty” icon opens as a new custom Form class (with the JFC extension) that you can save as a template for later use. See Using custom form class to create base forms.

- Right-click an untitled icon and choose either New Form or New Home Page Form from the context menu (the context menu is different, depending on which of the two untitled icons you select).
- Click and drag a form icon out of the IntraBuilder Explorer (or Windows Explorer) and onto the IntraBuilder desktop.
- Select a form (or an untitled icon) and click the Design button in the toolbar.
- Right-click an existing form (.jfm) file, and choose Design Form from the menu.

If you have a form on the desktop in Run mode, you can open the Form Designer by clicking the Design button on the toolbar.

New Custom Form, a base form to use as template for coordinated forms. New form icon: drag this to desktop to create a new form.

Design and Run modes

Related topics

IntraBuilder provides two modes for working with forms: a Design mode and a Run mode.

To switch between Design and Run modes, select a form in the IntraBuilder Explorer and click the toolbar buttons or right-click to choose from the context menu.

- In Design mode, you design the appearance and functionality of the form by placing controls on it. You also determine the behavior of the form and its controls by assigning properties.
- In Run mode, the form's controls become active. For example, you can enter data into a data-entry (Text) field. Any data you've linked to the form, such as records from a table, also becomes available at run time for viewing or editing.

The Form Designer menu

Related topics

There are two menus specific to Design mode:

Use the Layout menu to align components on the visual design surface. For information on the options in the Layout menu, see Form design surface. Or, point to a menu item with the mouse, and read its description in the status line at the bottom of the screen.

Use the Method menu to work with methods in the Method Editor. For information on the options in the Method menu, see Method Editor.

The Form Designer toolbar

Related topics

The Form Designer toolbar buttons are a convenient alternative to some menu commands:

If you hold the mouse pointer over a button, you'll see the name of the button appear just below it. As you pass the mouse pointer over each button, notice the hints on the status bar at the bottom of the screen.

The Form Designer toolbar is a floating panel set by default directly below IntraBuilder's menu bar. You can select two panels of buttons, Standard and Alignment, and drag either or both over to your work area where they float over other active windows for easy access.

To return the floating panels to the toolbar, drag the panels to the area where the toolbar formerly appeared, wait until the panel outline changes to the toolbar format, then release the mouse button. The panels snap back into the toolbar.

You can also click in the gray area around the buttons and reposition the panels sideways on the toolbar; this may be convenient on larger monitors.

The Form Designer context menu

Related topics

The context menu is a quick way to access the Properties dialog box for a selected object, to display design tools, and to perform common editing operations. To use the context menu, right-click anywhere in the Form Designer.

The Form Designer context menu contains the following options:

- Inspector—displays the object inspector that lets you view and edit the properties, events, and methods of IntraBuilder objects.
- Method Editor—displays a specialized text editor that lets you view, edit, and quickly create methods in the current form.
- Component Palette—show or hide the Component Palette, which offers a selection of user interface controls and data access components.
- Field Palette—show or hide the Field Palette, offering you a selection of the active fields available on the currently linked table.
- Toolbars—show or hide the toolbars.
- Cut, Copy and Paste—standard editing functions
- Form Designer Properties—displays Form Designer Properties dialog box, where you can turn the ruler on or off and set grid sizes and behavior.

Show or hide the Inspector, Method Editor, Component Palette, Field Palette, or toolbars. Cut or copy the selected control to the Clipboard; paste a control from the Clipboard to the form. Open the Form Designer Properties dialog box to customize designer tools.

Form Designer tools

Related topics

You can open any combination of tools and keep them open as you create your form, or minimize them on the desktop. Because Form Designer tools are windows, you can move and resize tool borders and switch focus among them. To organize all the tools on your desktop, choose the Window|Arrange Designer Windows command.

IntraBuilderYou can display these items by choosing them from the View menu item or by right-clicking and choosing them from the context menu.

- Form Design Surface

The visual design surface on which you will position text, graphics, and controls. The window, grid, and rulers are adjustable in the Form Designer Properties dialog box.

- Component Palette

A tabbed palette of standard user-interface controls and data access objects. You can place these components on the visual surface by double-clicking or by clicking and dragging to the surface. You can also create custom components and add them to this palette.

- Inspector

A three-tabbed palette that allows you to set properties, events, and methods for selected controls and database access objects.

- Field Palette

Displays active fields for a form's active Query objects. The fields are linked to views or rowsets of a table or database. You can drag these functional fields directly to the design surface without having to set the dataLink properties.

- Script Editor

Displays all the JavaScript generated by creating IntraBuilder forms, reports, and home pages in the experts and designers. When you run an IntraBuilder application and IntraBuilder detects a problem, you are alerted. When you click the Alert's Fix button, the Script Editor appears with the problem line indicated by the cursor position. Many properties of the Script Editor are customizable (Choose Properties|Editor Properties). To view or edit a script in the Script Editor, close the Form Designer, right-click a file and choose Edit As Script from the context menu.

- Method Editor

Lets you directly edit IntraBuilder's methods, a subset of the complete JavaScript code shown in the Script Editor. You can open the Method Editor from the View menu and keep it open at all times.

- Script Pad

The Script Pad is a handy statement-line window that lets you quickly experiment with single-line statements and expressions. You can instantly see the results in the Results pane. You can open the Script Pad from the View menu and keep it open at all times.

Form design surface

Related topics

The form window is a visual design surface on which you position text, graphics, controls (such as selection lists and check boxes) and data access objects (queries, stored procedures, databases, and sessions), moving and sizing objects as needed. You can finely adjust the ruler and grid settings by using the Form Designer Properties dialog box.

Here are the parts of the form window:

- The window itself. You can change the size and position of the window (thereby changing the form) by changing the values of these form properties: height, left, top, and width.
- The grid. The grid helps you to quickly align controls. The grid is a matrix of dots that you can adjust or remove. By default, the grid appears when you start the Form Designer and objects are constrained to line up along the grids (Snap To Grid).
- Vertical and horizontal rulers. The units on the ruler are based on the size of the current font. The vertical unit is equivalent to the height of the font. The horizontal unit is equivalent to the average width of the characters in the font. You can change all these settings. By default, the rulers appear when you start the Form Designer.

To customize how the grid and rulers appear and function on the design surface, choose Properties| Form Designer Properties. The options in the Form Designer Properties dialog box are described.

The Layout menu

Related topics

In addition to graphically sizing and arranging controls on the design surface by using the mouse, you can size and align selected objects by using the Layout menu commands.

- Align

The Align submenu offers you a choice of aligning the selected control to the Left, Right, Top, Bottom, Absolute or Relative Horizontal center, and Absolute or Relative Vertical Center.

- Size

The Size submenu lets you grow or shrink either the width or height of the selected control.

- Spacing

The Spacing submenu lets you adjust the horizontal and vertical space between items in a group.

- Set Scheme

Displays the Set Scheme dialog box which lets you save current font and color settings as a reusable scheme, which is useful for maintaining a look and feel over several pages of a form or across related applications. You choose a predefined scheme by selecting one in the Form Expert, or by choosing Layout|Set Scheme when the Form Designer is open

- **Delete** deletes the currently selected scheme.
- **Reset** restores the initial list of schemes.

Important Reset deletes any schemes you have defined, so use this command with caution.

- **Save As Scheme** lets you assign a name to the current set of font and color selections.
- **Apply** applies the current scheme to the current form.
- **OK** selects the current scheme for new forms and components.

Component Palette

Related topics

The Component Palette displays the controls and data access objects you can add to a form. You'll always see the Standard and Data Access tabbed pages. If you installed the SAMPLES directory, you'll also see Navigation and Update pages. Custom components designed by the developer are added to additional pages.

To open the Component Palette, do one of the following:

- Choose View|Component Palette from the menu.
- Right-click anywhere on the form window and choose Component Palette from the context menu

All tabs have a pointer button that lets you convert the cursor from a control icon back to a pointer. The Standard tab shows form controls. The Data Access tab shows database access objects required to connect to a table, group of tables, or to ensure record-locking.

Standard controls

Related topics

This table describes the standard user interface controls appearing on the Standard page of the Component Palette. After placing a component on a form, you link it to a table or rowset by setting its *dataLink* property. How to use each of these controls is described in separate subsections. All these controls and data access objects are accessible as JavaScript objects.

Standard controls on the Component Palette

Control	Use	Example/Explanation
Button	Perform a task with a single interaction.	Example: An OK button that, when clicked, performs the main action of the form, such as commits the changes the user has made to a table or moves to the next page or record.
CheckBox	Toggle between two choices of a logical value. Choose a number of options that are not mutually-exclusive.	Example: A *"*Credit OK*?"* box with which to enter *true* or *false* in a CREDIT_OK field of a table.
Radio	Select one choice among a group of mutually-exclusive possible values.	Example: A group of buttons labeled Credit, Cash, Check, Visa, and MC to choose among for entering only one of those values in a PAYMENT METHOD field of a table.
Rule	Horizontal divider organizes form layout, grouping related controls.	Example: Could separate address and name data entry controls from order data entry controls.
TextArea	View/edit text in a scrollable area of any size.	Displays a text file or block of text. Text exceeding the size of the box causes a scrollbar to appear. You can let users edit this text if you wish.
Select	Select one of any number of values from a drop-down list. User clicks the down-arrow button to display list.	Example: A data-entry area for entering a value for a Payment Method field of a table with a list of possible payment methods (in a related table) from which to choose. See Select: Creating drop-down selection lists for details on this control.
Text	Enter a single value, text or numbers, into a data-entry field. A frequently-used component for entering data of any kind.	Example: Data entry area for entering a value for a Customer field of a table. The value is posted to the underlying table when the user clicks a button that sends all the data added to the form.
Image	View a color graphic or photograph. Image may be a button to perform an action.	Display area for a bitmap image stored in a binary field, resource file, or graphic file. Link to another file, a navigation button, or execute a JavaScript program.
Reset	Clears any data the user has entered into the form's fields.	In any data-entry form, allows user to reconsider and revise, before posting the data to the table. A preset instance of the Button control.
Password	Enter a password to gain access to the Web server, the intranet, or to restricted databases.	A pre-set instance of the Text control that hides typed input; suitable for entering User ID or password. Could be used on login page for database-enabled part of an Intranet site.
JavaApplet	Runs a Java applet when the user opens the form.	Use this control to insert a place holder object for a Java applet. In the Inspector you set the properties of this object with a URL link to the applet code.
Hidden	The user does not see this object on the browser.	The Hidden object is an invisible data cache on the HTML form that is not visible to the user but allows you to store the result of an expression or a JavaScript code block, so that the result value can be returned to the server when the user clicks the Submit or Send button.
ActiveX	View and operate an ActiveX Internet-based application.	Use this control to insert a place holder object for an ActiveX applet. In the Inspector you set the properties of this object with a URL link to the applet code.
ListBox	Select one or more items from a	Examples: A list of files the user can open. A list of

	fixed-size list box.	non-exclusive options, such as toppings for pizza, from which the user may make multiple selections.
HTML	Heading, description, instruction, prompt, links, or other literal text set upon the background of the form.	Tagged text set upon the form background (a title or label). Text may be styled fonts with color (for browsers supporting embedded fonts) and may be tagged for full standard HTML functionality. Text can be active links to other pages or websites.

Data access objects

Related topics

Table 4.2 describes the four functional data access objects available from the Data Access page of the Component Palette. These objects provide live connections and session control to tables and databases. A form that accesses a table must have at least one query object on it. A stored procedure object that returns a rowset (as a query would) can replace the required explicit query object. Each data access object is described in detail in subsequent sections.

Data access objects

Object	Lets the user...	Explanation
Query	Run a query on any table, including local DBF and DB tables, as well as SQL, ODBC or other remote tables. Enables form components to display fields from the table on the form.	You must add a Query object containing the appropriate SQL statement to connect to any table or database. You can create a preset Query object by simply dragging a Table object from the IntraBuilder Explorer's Tables page to the Form design surface. REQUIRED
StoredProc	Run a stored procedure. This capability is only available when accessing tables on a server that supports stored procedures.	Place the StoredProc control on a form and link the control to a stored procedure. OPTIONAL
Database	Type in a login string to access a SQL database (or other group of tables identified by an alias).	Gives IntraBuilder forms access to SQL databases. To add connections to SQL databases or other multiple tables via a BDE alias, add a Database object to your form. OPTIONAL
Session	Session objects enable basic record-locking, so that multiple users do not modify the same record at the same time. Session objects also help to maintain security logins for local DBF or DB tables.	When you open a form, a default session is created, linking the form to the Borland Database Engine and connected tables. When you need separate threads for each user (to ensure record-locking), add a Session object to your form. A unique session number is assigned to track each user's connection to the table. If you need a session, be sure to add the session object first, next the database object, and finally the query object, in that order. In this way each database object is automatically assigned to the others. OPTIONAL

All the controls and data access objects described in the preceding table are JavaScript objects that may be customized, saved with special properties, event handlers and methods, and made available on the Component Palette for convenient reuse. See Custom components.

Navigation objects

Related topics

Table 4.3 describes the 10 functional navigation objects available from the Navigation page of the Component Palette. The Navigation page is a page of custom components that is delivered with the IntraBuilder/Samples directory. This page is present in IntraBuilder only if your installation includes the SAMPLES directory. These objects provide form controls that let users navigate through records in tables and databases.

Navigation objects

Object	What it is	What it does
Firstimage	An image-style first-record control.	Displays the first record in the table that is linked to the form.
Previousimage	An image-style previous-record control.	Displays the previous record in the table that is linked to the form.
Nextimage	An image-style next-record control.	Displays the next record in the table that is linked to the form.
Lastimage	An image-style last-record control.	Displays the last record in the table that is linked to the form.
Navigatehorizontalimage	A horizontal image-style set of navigation controls.	Contains the First, Next, Previous, and Last image controls arranged in a horizontal bar.
Navigateverticalimage	A vertical image-style set of navigation controls.	Contains the First, Next, Previous, and Last image controls arranged in a vertical bar.
Firstbutton	A button-style first-record control.	Displays the first record in the table that is linked to the form.
Previousbutton	A button-style previous-record control.	Displays the previous record in the table that is linked to the form.
Nextbutton	A button-style next-record control.	Displays the next record in the table that is linked to the form.
Lastbutton	A button-style last-record control.	Displays the last record in the table that is linked to the form.

All the controls and data access objects described in the preceding table are JavaScript objects that may be customized, saved with special properties, event handlers and methods, and made available on the Component Palette for convenient reuse. See Custom components.

Update objects

Related topics

Table 4.4 describes the 16 functional update objects available from the Update page of the Component Palette. Update objects present table operations to the form users. The Update page is a page of custom components that is delivered with the IntraBuilder/Samples directory. This page is present in IntraBuilder only if your installation includes the SAMPLES directory.

Update objects

Object	What it is	What it does
Addimage	An image-style add-record control.	Lets users put the table that is linked to the form into Append mode to enter a new record. Clicking Add again adds the new record to the table and keeps the table in Append mode.
Deleteimage	An image-style delete-record control.	Lets users delete the current row from the table that is linked to the form.
Editimage	An image-style edit record control.	Lets users edit the current row.
Saveimage	An image-style save-record control.	Lets users save the current row.
Abandonimage	An image-style abandon-changes control.	Lets users abandon any changes made to the current row and return to the last saved contents of the row.
Searchimage	An image-style search-records control.	Lets users go to the first row that matches the criteria. When the user clicks the Search control, the form goes blank. The user then types in the criteria for the search and clicks the Search control again.
Filterimage	An image-style filter-records control.	Lets users display records that meet a specific criteria. When the user clicks the Filter control, the form goes blank. The user then types in the criteria for the filter and clicks the Filter control again.
Updatehorizontalimage	A horizontal set of image-style update controls.	Contains the Addimage, Deleteimage, Saveimage, Abandonimage, Editimage, Searchimage and Filterimage controls arranged in a horizontal bar.
Updateverticalimage	A vertical set of image-style update controls on a form.	Contains the Addimage, Deleteimage, Saveimage, Abandonimage, Editimage, Searchimage and Filterimage controls arranged in a vertical bar.
Addbutton	A button-style add-record control.	Lets users put the table that is linked to the form into Append mode to enter a new record. Clicking Add again adds the new record to the table and keeps the table in Append mode.
Deletebutton	A button-style delete-record control.	Lets users delete the current row from the table that is linked to the form.
Editbutton	A button-style edit-record control.	Lets users edit the current row.
Savebutton	A button-style save-record control.	Lets users save the current row.
Abandonbutton	A button-style abandon-changes control.	Lets users abandon any changes made to the current row and return to the last saved contents of the row.
Searchbutton	A button-style search-records control.	Lets users go to the first row that matches the criteria. When the user clicks the Search control, the form goes blank. The

		user then types in the criteria for the search and clicks the Search control again.
Filterbutton	A button-style filter-records control.	Lets users display records that meet a specific criteria. When the user clicks the Filter control, the form goes blank. The user then types in the criteria for the filter and clicks the Filter control again.

All the controls and data access objects described in the preceding table are JavaScript objects that may be customized, saved with special properties, event handlers and methods, and made available on the Component Palette for convenient reuse. See Custom components.

Custom objects

Related topics

The Custom page of the Component Palette contains custom-built objects defined by the developer. The IntraBuilder SAMPLES directory contains several custom components that are displayed on the Navigation and Update pages of the Component Palette.

Custom components and their capabilities are completely up to the developer. Completely new components can be built from scratch. Existing components can be altered and saved as custom components.

For directions on adding custom components, see Adding components and to Custom Components.

Working with components

Related topics

When designing your form, consider the types of controls and data access objects you'll use to accomplish certain tasks. See Form design surface and Standard controls.

This section describes how to work with components: adding, resizing, aligning, and so on. For information on setting specific controls, see Setting components in Form Designer.

Adding components

Related topics

You add components to the form by selecting their icons from the Component Palette or from the Field Palette. (To have access to the Field Palette, you must have first placed an active Query object on the form.) It is often easier to use the Field Palette because its components are already linked to the fields of the table specified in the Query object.

You can quickly add an active Query object by simply dragging a table (from Windows Explorer or IntraBuilder Explorer) to the form design surface. See The Field Palette and Query object.

To add a component from the Component Palette,

1. Click the component on the palette to select it. When you pass the pointer over the Form Design Surface, the pointer turns into a representation of the selected object with a position pointer.
2. Drag the pointer across the form window until the component is the size you want, or click the form window without dragging to add a component in its default size.

If you uncheck the Revert To Pointer option in the Toolbars and Palettes dialog box, the pointer remains a control-image after you place the control. This lets you place multiple instances of this component without having to return to the Component d Palette to select the component each time. You can change the pointer back to its default behavior by clicking the Pointer control on the Component Palette.

Here are two alternative ways to add a component:

- Double-click the component in the Component Palette; it appears at a default position on the design surface.
- Drag the component from the Component Palette to the design surface.

Selecting components

Related topics

To work with a component once you've placed it on the form, first select it (or give it focus). Once you select a component, you can resize it, move it, or delete it. You can also change its properties.

To select a component, do one of the following:

- Click the component.
- Press Tab or Shift+Tab until it's selected.

When a component has focus, its *handles*—small, black squares around the periphery—are visible.

Selecting multiple components

Related topics

You can select several components at a time and work with them collectively. To do so, do one of the following:

- Select a single component, then Shift-click additional controls.
- Choose Edit|Select All to select all the components on a form at once, or Edit|Select Form to select just the form.
- Drag a selection border around the components you want to select:

1. Place the pointer just outside the area that contains all the components.
2. Hold the left mouse button and drag to the corner of the area diagonally opposite the starting point and release the mouse button.

   While dragging, a rectangular border forms from the starting point. The border disappears when you release the mouse button.

Handles appear around all the selected components.

To deselect multiple components, if the form is not selected, click anywhere on the form window outside any control.

Moving components

Related topics

To move a component, select it by clicking it. Keep the pointer within the borders of the component. Then, do one of the following:

- Drag the component to the position you want. As soon as you move the mouse, the pointer becomes a hand. This indicates you're moving the component.
- Press any of the arrow keys to move the component in the direction of the arrow.

To move a multiple selection of components, select and drag them or press any of the arrow keys. If you drag a multiple selection, you need to drag only one of the components. The other selected components move automatically.

If Snap To Grid is checked in the Form Designer Properties dialog box, then components align to the defined settings of the grid.

Cutting, copying, pasting, deleting

Related topics

Use the cut, copy, and paste controls the same way you would with text. Use the Edit menu, context menus, or toolbar buttons. To delete a selected component or multiple selection of components, choose Edit|Delete (or press Del).

Undoing and redoing

Related topics

You can undo operations on a form. Once you undo an operation, the previous action is available to Undo.

You can undo and redo values that you set in the Inspector. Once you undo a value, the Undo command on the Edit menu becomes Redo. Choose Redo to undo your last Undo operation.

To undo an operation, choose Edit|Undo (or press Ctrl+Z). To redo an operation, Choose Edit|Redo (or press Ctrl+Z).

Aligning components

Related topics

You can align components by using the Layout|Align menu commands or the corresponding toolbar buttons. The Layout|Align commands can be used to adjust the position of objects in relation to each other or in relation to the form.

- Align Left: moves the selected objects horizontally to the position of the leftmost selected object.
- Align Right: moves the selected objects horizontally to the position of the rightmost selected object
- Align Top: moves the selected objects vertically to the position of the highest selected object.
- Align Bottom: moves the selected objects vertically to the position of the lowest selected object.
- Align To Grid: moves each of the selected objects into alignment with the grid.
- Center Horizontally: aligns the horizontal centers of all selected objects.
- Center Vertically: aligns the vertical centers of all selected objects.
- Center Horizontally In Window: places the selected components in the absolute horizontal center of the Form Design window.
- Center Vertically in Window: places the selected components in the absolute vertical center of the Form Design window.

Resizing components

Related topics

To resize a component, select it and do one of the following:

- Place the pointer on one of its handles. When the pointer turns into a double-headed arrow, drag the handle to size the component the way you want.
- Press Shift+any arrow key to resize it in the direction of the arrow.

You cannot resize a multiple selection of components with the mouse; however, you can press Shift+an arrow key to resize a multiple selection in the direction of the arrow.

To conform the sizes of multiple objects, choose an option from the Layout|Size menu or the corresponding button in the Alignment toolbar:

- Grow To Largest Width: grows the selected objects to the width of the widest selected object.
- Shrink To Smallest Width: shrinks the selected objects to the width of the smallest selected object.
- Grow To Largest Height: grows the selected objects to the height of the largest selected object.
- Shrink To Smallest Height: shrinks the selected objects to the height of the smallest selected object.

Spacing components

Related topics

To distribute, or space, components, select the components, and choose an option from the Layout| Spacing menu:

- Make Equal Horizontal Spacing: distributes the selected objects horizontally to the width currently occupied by the objects.
- Increase Horizontal Spacing: increases the horizontal spacing between the selected objects.
- Decrease Horizontal Spacing: decreases the horizontal spacing between the selected objects.
- Make Equal Vertical Spacing: distributes the selected objects vertically to the height currently occupied by the objects.
- Increase Vertical Spacing: increases the vertical spacing between the selected objects.
- Decrease Vertical Spacing: decreases the vertical spacing between the selected objects.

Setting or changing properties

Related topics

You can change a component's properties in the Inspector. When you select a component in the form, the Inspector displays the component's properties.

When you have selected multiple components, you can change their properties simultaneously. After selecting one or more components, do one of the following:

- Right-click any component in the selected set and choose Inspector from the context menu.
- Choose View|Inspector.

When you change a property value or link code to an event for a multiple selection, the change affects all components in the selection. You cannot change methods for multiple selections.

Setting components in Form Designer

Related topics

This section gives instructions on using each of the controls and database access objects: how to add them to a form and implement basic or typical functionality.

Each type of control and data access object is a JavaScript object. IntraBuilder's extended JavaScript makes it possible to create varied and extended functionality for form components.

For details about setting an object's properties (editable in the Inspector), refer to that object in the latest version of the *Language Reference*.

Linking a form to tables

Related topics

Before you can link some components or set certain component properties, IntraBuilder may require at least one active Query object on the form. The Query object links the form to a table, making the table's fields available to the control objects on that form. You can add multiple Query objects, linking the form to multiple tables.

A Query object is created for you when you use the Form Expert. In the Form Designer you must add a Query object for each table you want to connect to your form.

The easiest way to add a live Query object is to click a table from the IntraBuilder Explorer's Tables page (or from the Windows Explorer) and drag it to your form. This creates a Query object on the form already linked to the selected table.

Alternatively, you can drag a Query object from the Database page of the Component Palette to the design surface and set its *sql* property for the desired table. See Query object.

Helpful suggestions:

- Begin by setting your Query objects on the form first (and activating them), so that the Field Palette displays a set of components already linked to each of the fields of the linked tables.
- Change the default names of new components to something more descriptive than "text1," "text2," and so on. This makes referring to these components in JavaScript much easier later on.

HTML: Creating titles, labels, and text

Related topics

You create titles (for an entire page or area) and labels (for individual objects) by using the HTML component. An HTML object generates HTML text, including links to pages or URLS, that appears fixed directly on the background surface of the form. You can place label text of any size anywhere on the form.

You can set the font, font style, and text color by using the Font Property Builder. The font style and color will appear on any browser that supports the font-embedding HTML extensions proposed by Microsoft.

You can apply HTML tags to any text string and check the results in the Text Property Builder sample pane. See Setting HTML tags.

To add label or title text to a form,

1. Click the HTML control on the Standard page of the Component Palette.
2. Click and drag the desired shape of the text area you want to create on the Design Surface. The resulting HTML object is selected for you.
3. With the HTML control selected, open the Inspector.
4. The first HTML control on a form is named, by default, “form.HTML1”.You might want to rename the object by expanding the Identification Properties and typing a new name in the *Name* property. Notice that the object name at the top of the Inspector changes.
5. Click the *Text* property, activating its entry field. Directly type the text you want to appear on the form. Alternatively, you can click the text tool and display the Text Property Builder. (See Setting HTML tags.)

The text you typed in the Inspector field now appears on the form.

Choosing fonts

Related topics

For browsers that support font-embedding, you can use any font and style to create colorful titles and labels. You can choose fonts for titles or labels in two ways using the Inspector:

- With the HTML control selected, expand the Font Properties heading in the Inspector. You can directly edit these properties, setting the font name and style.
- Or, click the tool button in the *FontName* property. The Font Property Builder appears.

Choose a font and style. Your selections are reflected in the HTML control on the form. Changes you make in this dialog box are reflected only in IntraBuilder, except for font styles. Font styles are reflected in both IntraBuilder and browsers. Select the settings you want, and click OK.

Note If you want to make other font changes that will affect the look of a form on a user's browser, use the Text Property Builder. You can access the Text Property Builder through the Inspector: with a text object selected, click the tool icon to the right of the text property.

Browsers that do not support font-embedding extensions to HTML will not display the fonts you have chosen.

You can use the Font Property Builder as an alternative to typing in the properties in the Inspector. This way you can view the font options and available sizes, then see how it will look in the sample pane.

Adding color to text

Related topics

If you want to add color to the text of an HTML control,

1. In the Inspector, expand the Visual Properties heading.
2. Select the *Color* property and click its tool button. The Color Property Builder appears.

Here you can select a basic color or custom blend a particular shade and even add that shade to the color palette for later reuse. Because intranets and the Web serve a multitude of computers with different color capabilities, it is usually a good idea to stick to 16 to 32 basic colors. These are more likely to display the way you want on most platforms.

You can create custom colors by clicking in the color field of the Color Property Builder. The top pane shows how text will appear in that color. When you find a shade you want to reuse, click “Add to Custom Colors” to add that shade to the Custom Color palette.

Setting HTML tags

Related topics

You can easily apply HTML tags, such as bold, italic, color or URL links, to text created in the HTML control.

To set HTML tags,

1. Click the tool button in the *Text* property. The Text Property Builder appears.
2. In the Text Property Builder, select the text you want to tag in the upper-right “Text without tags” box.
3. Use the controls at the left of the Text Property Builder to add basic font tags, URL or local file links, text color, and other (custom) HTML tags, as described below.

To apply basic HTML font style tags for selected text, click the Font tags:

Bold	Italic	Underline
Strikethrough	Subscript	Superscript

To make the selected text into a link to another HTML file or another URL, type the file path name or the URL in the URL Tag box and click Add. This encloses the selected text in <A HREF...> tags.

To make the selected text into a color, choose a preset color from the Color Tag drop-down selection list. Or click the Color tool button to display the Color Property Builder and create a custom color. When you have the color you want displayed in the Color Tag box, click the Add button.

To enclose the selected text in other HTML tags, select the tag from the Custom Tags drop-down selection list. If the HTML tag you want does not appear in this list, click the New button to display the Add Custom Tag dialog box.

To display a list of tags currently applied to the selected text, click on Tags at Current Position.

To remove the tag appearing in the box, click the Remove button. The bottom right area displays how the tagged text will appear in the HTML Web browsers.

To edit the custom tag appearing in the Custom Tags box, click the Edit button. This also displays the Add Custom Tag dialog box so you can modify the tag.

If you wish, you can type a name for the tag you want in the Tag Description box. Enter the start and end tags exactly as you would tag a text string in an HTML file, then click OK. The HTML code will be streamed from the server.

Click Font tag buttons to apply basic font style to selected text. Type filename or URL in URL tag to make selected text into a link. Click Color Tag to pick color for text. Click Custom Tags to choose other HTML tags. Click on Custom Tags|New to display Add Custom Tag. First select the desired text in the top right pane. You can apply different tags to different parts of a title or sentence. Click on Tags at Current Position to display list of tags currently applied to the selected text. Click the Remove button to remove the tag appearing in the box. The bottom right area displays how the tagged text will appear in the HTML Web browers.

Text: Creating data-entry fields

Related topics

Text controls create data-entry fields that accept any type of text input, including numeric values.You can link Text controls to any field type in a table. For example, you might use a Text control to create a data-entry field for a customer name, order number, date of sale, and so on. You will frequently use the Text control when creating data-entry forms.

To add a data-entry field to a form, click the Text control on the Component Palette, then click it where you want it on the form and drag to size it.

At this point, you can move, resize, and align the data-entry field (Text control) as you want. At a minimum, consider adding a descriptive label (HTML control) for the Text control.

Adding a label (HTML object) identifies the information contained in a Text control. At least one live Query object must be on the form, linking the form to a table so that the form object can be linked to a live field.

Linking a Text control to a field in a table

Related topics

To enable users to display and change field data when the form runs, link a Text control to a field in a table. For example, if you link the Name field in the Customer table to a Text control on a form, users can run the form and enter customer names into the data fields through that control. If you used the Form Expert to create the form, the data link is defined automatically.

In Form Designer, before you attempt to link a Text control you must link the form to a table. You do this by selecting a table from IntraBuilder Explorer's Tables page and dragging it onto your form. Alternatively, you can drag a Query object from the Component Palette to the form, and, using the Inspector, activate the query object, and set its *sql* property with a SQL statement that selects from the desired table. See Query object.

To link a Text control,

1. Add the Text control to the form.
2. In the Inspector, click the *dataLink* property tool.
3. From the Choose Field dialog box, select a Query object in the left pane (at minimum there must be one active Query object on a form) and a field of that table (or rowset) from the right pane. This field will be linked to your new text-entry field on the form.

When you complete the link, the field data in the current record appears in the Text control's data-entry field.

You can also constrain the types of data a user may enter into this field by setting the *template* property under the Edit Properties heading on the Inspector's Properties page.

Password: Creating a login

Related topics

You may want to add a password entry field to restrict access to your intranet form (and underlying database) to authorized users. A Password control is virtually the same as a data-entry field (Text control), except that when the user types in a value (on a form displayed on a browser) only asterisks appear, to hide the password entry from possible observers.

You might use the password object to clear access to a particular encrypted table, or you might create a user login on the first page of your form—a Login Page. In that case, you might title the page “Login please” and add a Name field (a regular Text control) and a Password (Password control). For a detailed example, see The TMD login form.

To add a password entry field,

1. Add the Password control to the form.
2. Add a Button control to the form.
3. With the Button control selected, in the Inspector, click the *onServerLoad* event.
4. In the Method Editor, provide JavaScript code for the button’s *onServerLoad* event that will submit the value entered into the Password field, according to the type of encrypted table you are accessing.

Select: Creating drop-down selection lists

Related topics

A Select control is an entry field with a drop-down list of possible entry values.
Select controls are useful when you want the field to offer a fixed set of possible values from which the user can choose only one value. For example, in a Select File dialog box, users can select an existing file from the list. Select controls are similar in appearance to “combo boxes” but differ in that users cannot enter a value in the box or edit a selected value.

You can use Select controls for string, numeric, date, and Boolean fields.

To add a Select control, click the Select control on the Standard page of the Component Palette, then place it where you want it on the form.

Linking a Select control to a field in a table

Related topics

You link a Select control to a field in a table so that users can conveniently enter a value in the field of a new record by selecting an item from the drop-down menu. For example, you might link a Select control to the City field in the Customer table, to make it easy for users to change or enter city names in the field. This would be appropriate when you are dealing with a specific area with a fixed number of cities.

To link a Select control to a table field,

1. Add a Select control to the form by double-clicking or dragging the Select control from the Component Palette.
2. Add a Query object for the table you want to use. In this example, select the CUSTOMER.DBF sample table (located in the SAMPLES directory) from the IntraBuilder Explorer's Tables page and drag it to the form design surface.

   The first Select object on a form is named: form.select1. When you drag the Customer.dbf table from the Tables tab and drop it on the form, a Query object linked to that table appears.

   3 Make sure the Select control is selected.
4. On the Inspector's Properties page, expand the Data Linkage Properties heading.
5. Select the *dataLink* property.
6. Click the tool button of the *dataLink* property. The Choose Field dialog box appears.

   The left pane lists the Query objects currently in this form. For a selected Query object, the right pane lists the linked table's fields. Click the field you want to connect to this Select control.

   7 The Choose Field dialog box shows the fields of tables associated with Query objects currently attached to this form. In this example query1 is linked to the CUSTOMER.DBF sample table. Click the field you want to connect to the Select control, in this case City. Click OK.

   You know this box is now linked to a field because the value for the CITY field in the first record of the table Customer.dbf is displayed. This value will disappear once the drop-down list is enabled and the user clicks the down arrow.

   8 Now the Select control shows the value of the selected field for the first record of the linked table. This tells you that the Select control is correctly linked; once deployed, the control will appear empty until the user picks an entry from the drop-down menu.

Now that you have linked the Select control to a table field, you are ready to create a list of data-entry items from which users can choose. This example continues in the next section.

Specifying selection items for the drop-down list

Related topics

You can display two types of data as selectable items in a drop-down list:

- **Array** displays elements of an array. An array is a special memory variable that can contain any data type (numeric, character, date, and expressions).
- File displays the names of files on the Web server in the drop-down list. This is useful for letting users select a file for operations such as opening files, deleting files, and so on. You can determine which files appear in the list by specifying a file skeleton. For example, a file skeleton of *.TXT includes only files with a TXT extension, and a file skeleton of *.* includes all files.

In the example used for the previous section on linking the Select control to the City field of the CUSTOMER.DBF sample table, the user is offered a limited set of data-entry options, certain cities within the firm's region. The easiest way to create this list of data-entry options is to create an array.

To create a drop-down list of data-entry options,

1. Make sure the Select control on the form design surface is selected.
2. On the Inspector's Properties page, select the *options* property. (You might need to expand the Data Linkage Properties heading to see this option.)
3. Click the tool button of the *options* property. The DataSource Property Builder appears.
4. Select Array in the Type list.
5. Click the tool in the DataSource list. The **Build Array dialog box appears.**
6. Type in the String field each item you want to appear in the drop-down menu of the Select control on your form. Press Return after each entry. (You can also enter expressions in the expression box so that the result of each expression is calculated and displayed in the drop-down list.) In this example we are entering cities on the island of Kauai. When you have finished entering elements into the array, click OK.
7. The DataSource Property Builder reappears, now with the array you just created in the DataSource field. Click OK.
8. Click the Run button in the toolbar to change from Form Design mode to Run mode. The form now appears much as it would appear on your users' browsers. The moment it appears, the current City value of the first record in the linked table will appear in the Select control, but vanishes as soon as you click the arrow button that drops down the list.

Click the down arrow and select a city from the drop down list. Your selection appears in the Select control. On a deployed form, the new value for City is submitted when the user clicks the submit button and a new record is created.

Note that the items displayed in a Select control are for display and selection only; they cannot be changed. For example, if you choose to display file names in the current directory, users can only select a file name from the list but cannot add a file.

If you will need to often change the selection options listed in the drop-down list, it may be more convenient to keep the items in a separate table rather than an array built into the application. In this way, when you want to change the items in the drop-down menu you need not edit your IntraBuilder application and redeploy it over the server; you just need to update the linked table. To implement this approach requires writing a few lines of JavaScript in the Method Editor. For this procedure, see Part II of this guide, "IntraBuilder JavaScripting."

Choose Array to enter a fixed list of selection items. To open the Build Array dialog box, click the tool button of the DataSource box.

Type each array element in the String box, clicking Add (or pressing Return) after each entry. To remove items, click them and then click Remove. The array elements you type will appear in the drop-down menu of the Select box on your form.

ListBox: Creating a multiple-selection list

Related topics

Use the ListBox control to display choices for the user in a fixed size box. For example, a selection list control on a form may provide a list of files users can choose to open, or a list of available stock numbers for inventory.

If you’re trying to decide whether to use a ListBox control (fixed-size, multiple selection) or a Select control (drop-down list), use the following guidelines:

- If you have room to show the open list, or want to let users select more than one item from a list, use a ListBox control. A ListBox control displays only existing field values; users cannot change those values.
- If you do not have room, use a Select control. A Select drop-down menu allows the selection of only one item.

To add a ListBox control, click the ListBox control on the Component Palette, then place it where you want it on the form.

Specifying selection items for a ListBox control

Related topics

You can display two types of data that can be selected in a ListBox control:

- Elements in an array
- Files in the current directory

These items are for display and selection only; they cannot be edited by the user. For example, if the ListBox control displays file names in the current directory, users can only select but not change a file name from that list.

By default, the Form Designer allows one selection per ListBox control. However, you can permit multiple selections in a list by setting the *multiple* property to *true*. For example, if a dialog box contains a list of files for copying, the multiple-selection capability lets users select and copy multiple files at a time.

To specify list items to appear in a ListBox control,

1 Select the ListBox control on your form.

2 On the Inspector's Properties page, expand the Data Linkage Properties heading.

3 If you want users to be able to select more than one item, set the *multiple* property to *true*.

4 Click the *options* property and click its tool. The DataSource Property Builder appears.

5 Select Array in the Type list box.

6 Click the tool button of the Data Source box. The **Build Array dialog box appears.**

7 Type in the String box each item you want to appear in the ListBox box on your form. Press Return after each entry. You can also enter expressions in the expression box. The result of each expression will be calculated and displayed in the ListBox control. Click OK when you have finished entering string and expression array elements.

The array elements you added in the Build Array dialog box now appear in the Data Source list box.

8 Now the array elements you added appear in the Data Source box of the DataSource Property Builder. Click OK.

9 The ListBox control on your form now displays the array you created. Users will be able to any or all of the items in this list.

The array elements you added in the Build Array dialog box now appear in the finished ListBox control.

The Type list offers you a choice of Array or File. Click the tool button of the Data Source list to display the Build Array dialog box to help you quickly add elements to an array.

Type each array element in the String box, clicking Add (or pressing Return) after each entry. To remove items, click them and then click Remove. The array elements you type will appear in the ListBox box on your form.

Button

Related topics

You can add HTML-standard buttons, give them names and assign any action to be performed when a button is clicked. Buttons are used for navigating (moving back and forth through a series of records), running queries, or as links to other forms, reports, Web pages, or Web sites.

The easiest way to add the typical buttons for navigation, editing, and querying, is to create a form by using the Form Expert. The Form Expert presents a page that lets you choose a number of preset Button controls, including alternative image-style buttons, icons.

To create a button,

1. Add a Button control to the form.
2. In the Inspector, click the *Text* property and type in a name to appear on the button.
3. On the Inspector's Events page, click the *onServerClick* event's tool button. This displays the Method Editor with a function format.
4. In the Method Editor, enter JavaScript for the task you want the server to perform when the user clicks this Button control. To create a Next button that displays the next record when clicked, use this JavaScript:

```
function button1_onServerClick()
{
if (!this.form.rowset.next())
  this.form.rowset.next(-1);
}
```

Note The *next* method moves the row pointer. If it returns *false*, the end of the set has been reached and the rowset should be restored to its original position.

Most often you will want a button that simply saves the user's input, posting it to the linked table. This button is created automatically when you use the Form Expert.

For example, in Form Designer, to create a button to allow the user to add a new row to the table (such as "add New Employee"), you would enter in the Button control's *onServerClick* event:

```
{;this.form.rowset.beginAppend()}
```

To create an Edit button to allow the user to edit the fields of an existing row, you would enter in the Button control's *onServerClick* event:

```
{;this.form.rowset.beginEdit()}
```

To create a Save or Post button to allow the user to save all changes to the table, you would enter in the Button control's *onServerClick* event:

```
{;this.form.rowset.save()}
```

In some cases you might want to update several fields (such as date and time of user inquiry) along with the Button control's click event. For example, see the method for the Submit button in the sample application Guestbook. To view the method, in Design mode, open the Method Editor and choose the method for the event SubmitButton_*onServerClick*.

See Custom forms and components and Database access from forms for advanced Button control methods.

If you want fancy or graphical buttons, you can use the Image component instead, setting its onServerClick event to fire the same method as would a button. (The Form Expert offer pre-designed graphical icons for basic Button controls.)

Reset Button: Clearing a form

Related topics

The Reset Button control is an HTML-standard button that has been preset to provide the functionality of a reset button in an online form. A reset button allows the user to revert a form to its blank or initial settings. This lets users correct errors, revise, and start over when entering data into forms displayed on their browsers. By default, the Reset Button control affects only the browser display by resetting the HTML form; it does not send modifications to the server.

All you have to do is add the Reset Button control to your form.

CheckBox: Creating check boxes for logical data

Related topics

Check boxes accept selections between two opposite conditions, such as yes or no, on or off, and true or false. Typically, you want the *Checked* property of the CheckBox control to indicate true. For example, you might want to offer a check box for the question “Exempt?” If checked, the meaning is true, that is, exempt. If not checked the meaning if false, that is, not exempt.

Check boxes are also called *toggles* because you can switch between two states: checked (yes, on, true, selected) or unchecked (no, off, false, cleared). You can link check boxes to logical fields in a table.

To add a check box to a form, click the CheckBox control on the Component Palette, then place it where you want it on the form.

Any check box may be checked or unchecked. Check boxes are independent.

Note

Check boxes, while similar in appearance to radio buttons, work very differently during data entry. Radio buttons operate as a group; when one radio button is selected, all others in the group are unselected. Check boxes, however, function independently of each other even when visually grouped onscreen; checking one check box doesn’t uncheck others.

Set the Checked property to set whether the check box is checked (true) or unchecked (false). It’s unchecked by default. Set the Text property to change its label text to a more descriptive name. Change the text font by setting the Font properties.

Linking a CheckBox control to a logical field

Related topics

You can link a CheckBox control to a logical field in a table so that users can display and change the field when the form runs. For example, if you link a check box to the CREDIT_OK field in the Customer sample table, users can run the form and change the logical value (yes or no, true or false). If you used the Form Expert to create the control, the link is defined for you automatically. Otherwise, before you can link a new CheckBox control, you must first add to your form an active Query object linked to the desired table. The quickest way to do this is to drag a table from the IntraBuilder Explorer's Tables page to the form design surface. (See Query object.)

To link a check box,

1. Add the CheckBox control to the form.
2. In the Inspector, click the tool button in the *dataLink* property. The Choose Fields dialog box appears.
3. Select a field to connect to this CheckBox control, or enter a JavaScript statement in the text box next to the *dataLink* property, such as:

   ```
   parent.query1.rowset.fields["CREDIT_OK"]
   ```

When you complete the link, the field value in the current record (true or *false*) appears in the check box.

Radio: Creating a group of radio buttons

Related topics

Radio buttons allow users to select a single choice from among a set of alternatives. For example, payment terms with a customer might be a 15-day, 30-day, 45-day, or 60-day term, depending on the terms of sale. Only one of these in the set of alternatives applies to a particular sale.

Radio buttons by definition are part of a group of at least two or more. If you select one radio button in the group, all other radio buttons are unselected. The selected option becomes the value that applies to the linked field.

To create a group of radio buttons,

- Place each Radio Button control.
- Group the Radio Button controls.
- Link each Radio Button control in the group to the same table field.
- Assign a descriptive label to each Radio Button control; when the user selects a radio button, its label is the value that is entered into the table field.

To add a radio button, drag the Radio Button control from the Component Palette and drop it where you want it on the form. Use the Inspector to set its value, type descriptive text for the label, or even assign a color to the label text.

Grouping radio buttons

Related topics

To create a group of radio buttons, you must add each Radio Button control in the group consecutively. After you add the first Radio Button control of a group, set its *groupName* property. For all subsequent Radio Button controls in the group, set their *groupName* property to match. To start a second group, set a new *groupName* property of the first Radio Button control in the next group.

To group a set of radio buttons, for each button,

1 Select the Radio Button control.

2 On the Inspector's Properties page, expand the Identification Properties heading.

3 Type a value in the *groupName* field (such as "Terms").

Linking radio buttons to a field in a table

Related topics

You link all the radio buttons in a group to the same table field so that when the user chooses a radio button, its value is inserted into the field. For example, when the user selects the “Net 30” radio button, its text value “Net 30” is inserted in the Order table’s field “Payment Terms.” (In the Orders sample table, this field is named PAY_METHOD.)

To link a group of radio buttons to a table field,

1. Select all the Radio Button controls.
2. In the Inspector, click the *dataLink* property.
3. Click the tool button in the *dataLink* property.
4. From the Choose Field dialog box, select a field from those available in the table of the selected active query. Or, enter a JavaScript statement in the text box next to the *dataLink* property.

Specifying values to enter in the table field

Related topics

When the user chooses a radio button, its value is entered into the linked table field. You specify the value to enter by setting the *Text* property. Unlike the *Text* property in other controls, the *Text* value you set for a Radio Button control has two purposes—it is the descriptive label and the actual value entered in the table field.

For example, if all the Radio Button controls in a group are linked to the PAYMENT field, choosing the Cash radio button (Figure 4.39 on page 4-37) inserts Cash into the field.

Note The *value* property of a Radio Button control is true or *false* to indicate if it is selected. The *value* property is not equal to the value of the linked field.

Rule: Dividing parts of a form page

Related topics

You can divide sections of text and controls by adding a horizontal rule to your form. The Rule object generates the HTML code <HR>, a variable width horizontal line. At thicker settings HTML rules may appear embossed on the surface of a Web page.

To change the thickness of the rule,

1. Expand the Visual Properties heading on the Inspector's Properties page.
2. Click the *size* property.
3. Click the spinbox to raise or lower the number from 1. You can set the rule size quite high, but usually a value between 1 and 10 is most appropriate.

As an alternative to the standard HTML horizontal rule you can use an Image object to import a GIF graphic file.

TextArea: Displaying or editing extensive text

Related topics

The TextArea control displays and manages long text strings that can vary in length. You can link TextArea controls to text fields, memo fields, and ASCII text files. For example, you might want to use a TextArea control to display a 120-byte character field or to display memo notes about each customer on a data-entry form. Similarly, you might want to let users see the contents of a README.TXT file.

To add a TextArea control, click the TextArea control on the Component Palette, then drag it to the size you want on the form design surface.

Linking the TextArea control to a table field or text file

Related topics

Link the TextArea control to a character or memo field in a table so that users can display and change field data when they run the form. You can also link a TextArea control to an external text file to display and change its contents.

To link a TextArea control to data,

1. On the Inspector's Properties page, click the Tool button of the *dataLink* property to display the DataLink Property Builder.For a character or memo field, select Field from the Type list. For a text file, select File from the Type list.
2. Click the Tool button in the DataLink box.
3. Select a field or file in the DataLink Property Builder.
4. Click OK to close the DataLink Property Builder.

The name of the field or file appears in the *dataLink* property text box, and the linked data appears in the TextArea control.

Making the TextArea control read-only

Related topics

By default, the Form Designer lets users edit the contents of a TextArea control. However, you can make a TextArea control read-only to ensure that users cannot change its contents. You might want to do this when displaying an error message in a dialog box, or when users want to display the AUTOEXEC.BAT file without making any changes.

To make a selected TextArea control read-only,

1 On the Inspector's Properties page, expand the Edit Properties heading.

2 Set the *ReadOnly* property to *true*.

Image: Adding pictures to forms

Related topics

Most browsers can easily display GIF and JPG compressed images. You can import images (binary files) from image tables or picture galleries and you can place color image bitmaps, including artwork, photographs, or screen shots, of any size on IntraBuilder forms. You can also use graphics as dividers (an alternative to standard HTML horizontal rules) and as custom buttons that, when clicked, provide any functionality that you might otherwise assign to a standard HTML button.

The easiest way to add a picture to a form is to simply drag the graphic file from the Explorer (either IntraBuilder or Windows) and drop it on the form design surface. This creates an Image object that imports the graphic file to the form; the image appears on the form.

Important Don't forget to include the graphics file with your other IntraBuilder application files when you deploy it over the Web.

Sometimes, however, you may want to establish and arrange image locations before you have obtained or selected the images that will appear there. Then just add Image objects from the Component Palette.

To add an Image object, click the Image object on the Component Palette, then drag it to the size you want on the form design surface.

The Image object is a blank placeholder until you link the Image object to a table's binary file or to GIF or JPG image file.

To link an Image object to a graphic file,

1. On the Inspector's Properties page, click the Tool button of the *dataSource* property to display the DataSource Property Builder.
2. **To place a graphic image file,**
    1. Select Filename from the Location list box.
    2. Click the Tool button in the Image box. The Choose Image dialog box appears.
    3. **Select an image file in the Choose Image dialog box.**
    4. Click Open.
3. To place a binary field image (such as an image in a linked table or database) you must have an active Query object on the form linked to the table or database containing images.
    1. On the Inspector's Properties page, click the tool button of the dataSource property to display the DataSource Property Builder. Here you choose a binary file from a linked table.
    2. Select Binary from the Location list box.
    3. Click the Tool button in the Image box.
    4. In the Choose Field dialog box, select the binary image field you want. These are displayed only for the currently linked tables or databases.
    5. Click OK to close the Choose Field dialog box.
4. Click OK to close the DataSource Property Builder.

The picture contained in the graphic image file (or the binary field from a linked table) appears on the form. You can still move or resize it.

Image types other than GIF or JPG are automatically converted when the IntraBuilder server sends the HTML code stream to the user's browser.

JavaApplet: Accessing Java applets

Related topics

The JavaApplet object provides a resizable area on your form page in which to run a Java applet.

A Java applet is a program written in Sun Microsystem's Java,[a] a multi-threaded, object-oriented, platform-independent programming language.

You can place a Java applet in an HTML page, much as you would an image. When a user with Java-compatible browser opens a page that contains a Java applet, the applet's code is transferred from its residence anywhere on the Internet to the user's computer and executed by the browser.

Java applet resources (including their classes) are loaded relative to the document-URL (or <base> tag, if defined). You use the *codeBase* property to change this default behavior. If defined, the *codeBase* property specifies a different location to find applet resources.

The *codeBase* property value can be an absolute or relative URL. The absolute URL is used as-is without modification and is unaffected by the document's <base> tag. If the *codeBase* property is relative, then it is relative to the document-URL (or <base> tag, if defined).

You should first examine the applet code to find out how much room it needs
so that you can size the JavaApplet component on your form design surface to accommodate it.

To add a JavaApplet component,

1. Click the JavaApplet icon in the Component Palette and drag its rectangle to the approximate size required.
2. On the Inspector's Properties page, expand the Position Properties heading. Set the *width* and *height* properties to match the width and height definitions within the applet code. In the preceding applet example, width=300 and height=50.
3. On the Inspector's Properties page, click the *codeBase* property and enter an absolute or relative URL to the Java applet. The absolute URL for Nervous Text might be

   ```
   http://java.sun.com/java.sun.com/applets/NervousText
   ```
4. On the Inspector's Properties page, click the *code* property. Set it to the name of the actual access function of the Java applet inside the codebase. In our example, the *code* property would be

   ```
   NervousText.class
   ```

   Like JavaScript, Java is case-sensitive. The code property must be capitalized as it is defined in the Java applet.
5. On the Inspector's Properties page, click the params property and click the Tool button to use the Params Property Builder to set any parameters required by the Java applet.

ActiveX

Related topics

ActiveX[a] (formerly OCX) is Microsoft's component technology that can be embedded in HTML pages, much like Java applets. ActiveX is not cross-platform, so ActiveX-enhanced Web pages run on Microsoft Windows 95/NT browsers only. ActiveX support is built-in to Microsoft's Internet Explorer; a plug-in is available to run ActiveX components in Netscape Navigator.

To add an ActiveX object,

1. Click the ActiveX control in the Component Palette and drag its rectangle to the approximate size required.
2. On the Inspector's Properties page, expand the Position Properties heading. Set the *width* and *height* properties to match the width and height definitions within the applet code.
3. On the Inspector's Properties page, click the *codeBase* property and enter an absolute or relative URL to the ActiveX control. For example

   `http://activex.microsoft.com/controls/iexplorer/ielabel.ocx`
4. On the Inspector's Properties page, click the *classId* property and enter the control's ID string. For example

   `clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2`
5. On the Inspector's Properties page, click the params property and click the Tool button to use the Params Property Builder to set any parameters required by the ActiveX control.

Hidden

Related topics

The Hidden component is a JavaScript programmer's object for storing a value in an HTML document and returning it to the server, without exposing the transaction to the user. The value stored could be anything—the result of an expression. It will be returned to the server (along with all object values) when the user clicks the button (Submit, Send, Save, or whatever) that posts the form data to the server. For details on using the Hidden object, see Hidden components.

Query object

Related topics

The Query object provides a form with access to a table. At least one Query object is required in a form to link controls (via the *dataLink* property) to a table's fields.

A Query object contains an SQL statement (in its *sql* property) and the rowset (group of records) that results from it. A rowset represents some or all the rows (records) of a table or group of related tables. Each query generates only one rowset, but you can add multiple Query objects to a form to use multiple rowsets from the same table, or for different tables. Using multiple Query objects also allows you to take advantage of IntraBuilder's built-in master-detail linking.

The Query object's rowset (referenced in its *rowset* property) also maintains the current row and navigation, buffering, and filtering methods. All navigation methods for getting around in tables depend on the query's rowset. See Database access from forms for detailed guidance on setting rowset properties for navigation and filtering.

When you create a form by using the Form Expert, a Query object is automatically created for the specific tables you associated with the form. When you look at the new form in Design mode you see the Query object on the design surface (it is invisible in Run mode). In the Inspector you see that its *sql* property is set to the SQL statement SELECT * FROM *tablename*. This statement selects all the records in the associated table (that you picked in Step 1 in the Form Expert).

For SQL-server tables (table types other than DB or DBF) you must add a Database object to the form to access the SQL-server before adding the Query object. Database objects are described next.

Important It is easiest to create a preset Query object by simply dragging a table from the IntraBuilder Explorer's Tables page (or from the Windows Explorer) and dropping it on the form design surface. This gives you an active Query object already linked to the table, with a Database object if necessary.

Database object

Related topics

A Database object gives a form access to SQL-server databases by means of a Borland Database Engine (BDE) alias. You must use a BDE alias to access SQL-server and ODBC databases. Therefore to access tables in those databases, you must use a Database object in addition to a Query object.

You may also create a BDE alias for a directory of standard DB and DBF tables. Using a BDE alias for DB and DBF tables makes it easier to move the tables to another directory; only the alias in the BDE configuration needs to be updated, and not the source code for all the forms and reports. A BDE alias also makes it easier to change the table type from DB or DBF, which you might use during prototyping, to an SQL-server database.

When accessing tables through a BDE alias, follow this general procedure:

1. Add a new Session object to your form if necessary (as described next).
2. Add a new Database object to your form. It is automatically linked to the Session object already on the form.
3. Set the *databaseName* property to the name of the BDE alias.
4. Add a new Query object to your form. It is automatically linked to the Session and Database objects already on the form.
5. Set the Query object's *sql* property and make the Query object active.

Important To access a table through a BDE alias, it's easiest to Look In that alias in the Tables page of the IntraBuilder Explorer, then drag a table from the IntraBuilder Explorer and drop it on the form design surface. This gives you both the Database object connected to that BDE alias, and an active Query object linked to the table.

Session object

Related topics

A Session object provides a separate connection to a table or database. Sessions are used primarily for DB and DBF table security. Multiple users can each have their own session, so that different users can be logged in with different levels of access, or they may share a single session, so that all users have the same level of access.

If you are going to use a Session object, be sure to add it first, then the Database object, and finally the Query object, in that order. In this way each Database object is automatically assigned to the others. (See the procedure in Database object.)

Each session contains one or more Database objects. A session always contains a default Database object intended to directly access standard tables. You must create new Database objects to use tables through a BDE alias. Once you set the BDE alias, d activate the database object, and login if necessary, you have access to that database's tables. You may also log transactions or buffer updates to each database to allow you to rollback, abandon, or post changes as desired.

The Inspector

Related topics

The Inspector displays the properties of the form components (both controls and data access objects), as well as their events and methods. When you work with object properties, events, and methods, you should understand basic object-oriented program planning and the basics of JavaScript.

To open the Inspector (or make it active), do one of the following:

- Choose View|Inspector from the menu.
- Right-click anywhere on the form and choose Inspector from the context menu.

The Inspector is divided into three tabbed pages; Properties, Events, and Methods.

The name of the current object appears in the selection list box at the top of the Inspector. Click the Down arrow of the Select box to display the properties, events, and methods for each object.

Properties tab

Related topics

The Inspector's Properties page displays the properties of the current object. The right column shows the current value for each property.

You can set a property value in any of the following ways:

- Type the value into the column to the right of the property.
  Some property values appear on a character-by-character basis as you type them. For example, when you type a value for the *Text* property, each character appears in the object as you type it in the Inspector.

Note Yellow highlighting of an entry means "Not yet committed" or "Not yet evaluated." Press Enter to commit a change.

- Double-click the right column to rotate through a list of properties or to toggle logical values.
- Click the tool button that appears to the right of the property value when that property has focus in the Inspector. Tools are not available for every property.
  The tool button might produce a menu of available choices or a property builder in which you can choose a value. For example, you can display the *Color* property builder to set the color for an object.

Categorical or alphabetical display

Related topics

You can display properties categorically or alphabetically. When properties appear categorically, the Properties page lists several category names. A category name always has a plus (+) or minus (–) sign to indicate a list is available beneath it. A plus (+) sign indicates that the list is closed; a minus (–) sign indicates that the list is expanded. To toggle between open and closed, double-click the category name or press Enter (or the + or – keys on the numeric keypad).

When you open this category list, the plus sign becomes a minus sign, and the list appears expanded.

To open all category lists, press Ctrl and + on the numeric keypad. To close all category lists, press Ctrl and – on the numeric keypad.

If you prefer to have the Inspector display properties alphabetically,

1. Choose Properties|Desktop Properties.
2. Select the Application tab of the dialog box.
3. Uncheck the Inspector Outline check box

Events tab

Related topics

The Inspector's Events page displays the events to which the current object can respond. When you select an event, its value area becomes a text box with a Tool button.

To specify what will happen when an event occurs, you can do one of the following:

- Type a code block into the text box for the event.
- Write a method to link to the event

To write a code block and link it to an event,

1. On the form, click the object to whose event you want to attach the code.
2. Open the Inspector.
3. Click the Events tab.
4. Click the event to which you want to link the code block. The insertion point appears in the text box to the right of the event name.
5. Type the code block into that text box or click the tool button to display the Method Editor with the format of a new method ready to go.

IntraBuilderYou can also link and unlink events by using the Method menu from within the Method Editor. See Method Editor.

Methods tab

Related topics

The Inspector's Methods page displays the current object's built-in methods, that is, the methods pre-defined for the component. You can call these methods in methods you create with the Method Editor. Methods you create in the Method Editor can be inspected on the Methods page.

Note A function inside a class is a "method." The keyword for method is "function."

The Field Palette

Related topics

The Field Palette displays fields for each active query object linked to an existing table. You use the Field Palette as a convenient source for components already linked to a table. This saves you the work of manually dragging new “blank” components from the Component Palette, then activating setting the *dataLink* property for each component in the Inspector. Instead, you just drag the activated, already *dataLink*ed components from the Field Palette to the design surface of your form.

If no table is open (that is, if no active Query object exists on the form), the Field Palette is empty, showing only the Pointer button. When you begin to design a data-entry form in Form Designer, you must first add a Query object and set its properties. It is easiest to drag a table from the Tables page of the IntraBuilder Explorer (or from the Windows Explorer) to the design surface. This creates a Query object that selects all the records in that table, linking them to the form.

Once an active Query object exists on the form (its *active* property set to *true*), its fields appear on the Field Palette as active, linked components. The type of the component depends on the data type of the field. An independent Boolean field would appear on the Field Palette as a CheckBox control. Most fields are represented as a Text object (text entry fields).

If more than one table is open (more than one Query object exists on the form), the Field Palette displays a tabbed dialog box containing the fields in each table.

To open the Field Palette, right-click anywhere in the Form Designer and choose Field Palette from the context menu.

Script Editor

Related topics

IntraBuilder offers four text-editing tools for working with JavaScript:

- The Script Editor

The main window for editing JavaScript. Script Editor displays *all* the code in a selected file. To view or edit a script in the Script Editor, close the Form Designer, right-click on a file and choose Edit As Script from the context menu. Your work is saved when you close IntraBuilder.

- The Text Editor

Virtually identical to the Script Editor, this editor displays text when you wish to edit an HTML file or view a README.

- Method Editor

A specialized window that makes creating new methods easier and lets you quickly browse and analyze just the methods in a script. You can keep the Method Editor open and use it while you are working in the Form Designer.

- Script Pad

A two-paned statement-line interface that lets you quickly and temporarily experiment with JavaScript statements and expressions, instantly viewing results. You can use the Script Pad freely at any time while working in the Form Designer. The work in the Script Pad is not saved.

All four text tools are highly customizable, enabling you to define a number of properties—such as auto-indent, smart tabs, and syntax highlighting—that can make working with code both more comfortable and more efficient.

To set usability properties for IntraBuilder's three text editing tools, select the tool you want to customize and from the Properties menu choose Properties for that tool. The properties define operational preferences for all four editors at once. See Customizing for details about setting these properties.

To open the Script Editor,

1. Close the Form Designer before attempting to use the Script Editor.
2. From the IntraBuilder Explorer's Forms page, select a form file (with the JFM extension).
3. Right-click and from the context menu choose Edit as Script.

The Script Editor opens the selected form.

Running and debugging scripts

Related topics

You can save and run or debug your scripts as you write them. To run a script, do any of the following:

- Double-click the file icon of any JavaScript-based file in IntraBuilder Explorer.
- Select the JS file icon, and press F2 or click the Run button in the toolbar.
- Right-click the script file icon and choose Run Script from the context menu.
- In the Script Pad input pane, type

```
_sys.scripts.run("script_name.js")
```

Fixing script errors

Related topics

When you run a script that contains an error, IntraBuilder displays the Alert dialog box. This describes the problem and its location. Click the Fix button. This displays the Script Editor with the problem line pinpointed.

This table describes the options of the buttons at the bottom of the Script Alert dialog box.

Script Alert dialog box buttons

Button	Description
Cancel	Stops execution of the script and returns you to the Script Editor or the Explorer.
Fix	Opens the form or script file in the Script Editor, with the insertion point on the line containing the error.
Ignore	Disregards the error and continues execution of the script.

Script Pad

Related topics

The Script Pad window is used to directly execute one-line IntraBuilder statements. It is a handy scratch pad for testing simple expressions and immediately seeing the results in the Results pane.

Note The Script Pad is for temporary work only. Use the Script Editor if you want to save your work.

To use your own functions in the Script Pad, you must first load them:

```
_sys.scripts.load(filename)
```

You can access the Script Pad from the View menu.

The Script Pad has two panes, as shown in Figure 4.55.

Panes have specific functions:

- The input pane is where you enter JavaScript statements.
- The input pane echoes your actions in the IntraBuilder interface, keeping a history of the statements you've executed, not just in the Form Designer, but throughout IntraBuilder.
- The results pane is where your statement output appears, unless your statements create or call separate windows. It can be used with _sys.scriptOut.writeln to debug applications.

To temporarily clear the contents of the input pane, choose Edit|Select All and press Del or close the window and select View|Script Pad. The next time you execute a statement in the window, the full history list is restored.

To clear the results pane permanently, choose Edit|Clear All Results.

Typing and executing statements

Related topics

To execute a statement, type it in the input pane and press Enter. You can also click the Execute Selection button on the toolbar or choose Edit|Execute Selection. You can delete statements like any other text. The statements you enter in Script Pad remain there until you close the window or exit IntraBuilder.

The maximum number of characters per line is 300. The maximum number of lines the input pane can hold is limited by virtual memory.

The statement line defaults to insert mode, as indicated in the status bar. To switch between insert and overwrite modes, press the Ins key.

Executing multiline statements

Related topics

Because pressing Enter executes the statement line, you must press Ctrl+Enter to type more than one line into the Script Pad. Alternatively, you can use the down-arrow key.

In addition to typing multiple statement lines, you can paste lines of statement text from another source. You can also execute a block of statement lines, provided the block does not contain nested structures or methods.

To execute more than one line of text in the input pane, select the lines with the mouse or use Shift and the arrow keys. Press Enter, click the Run button on the toolbar, or choose Edit|Execute Selection.

Reusing statements

Related topics

To reuse statements you've already entered in the input pane,

1. Scroll the window, if necessary, to display the statements you want.
2. Click the statement line you want, or select a block of statements.
3. Execute the statement (or statements) by pressing Enter, clicking the Run button, or choosing Edit| Execute Selection.

Editing in the Script Pad

Related topics

Edit text in the input pane as you would in a text editor, using standard editing keys such as Backspace and Del, and the Edit menu commands, Cut, Copy, and Paste. Use the Edit|Search commands to search and replace text in the input pane.

The statement line is the line in the input pane containing the insertion point.

Copy text from the results pane and paste it anywhere you can paste text. Block selection in the results pane is unique in that you can select columns of text within a range of lines, without having to select entire lines.

You can copy statement syntax or sample code from the Help system's Language Reference directly into the input pane.

You can use statements from a script file by opening the file, copying the statements, and pasting them into the Script Pad. After the statements are in the Script Pad, you can test or modify them. The sample files provided with IntraBuilder are a good source of working statements.

Saving statements into scripts

Related topics

If the input pane contains code you want to use again, you can copy and paste it into a new script file or insert it into an existing script file.

You can also mark a block and choose Edit|Copy To File. IntraBuilder displays the Copy To File dialog box so you can name the new file for the selected text. By default, the file has a JS extension, but you can change it to another extension. If you don’t mark a block before you choose Edit|Copy To File, the entire contents of the Script Pad are selected.

Method Editor

Related topics

The Method Editor helps you write and structure the code that defines the form controls and their behavior. The Method Editor is an object-oriented programming tool and requires some familiarity with JavaScript. Be sure to read the onscreen documentation in IntraBuilder Help to gain an understanding of the concepts you need to know to work with this tool.

Opening the Method Editor

Related topics

To open the Method Editor without creating a new method, do one of the following:

- Choose View|Method Editor.
- Right-click anywhere on the form and choose Method Editor from the context menu.

If the form already has methods, the first method in the method list is current here and in the Method Editor window. If the form doesn't have methods, the "Header" section is selected.

To open the Method Editor and create a new method, in the Inspector, click the Events tab, choose an event, then click the tool button to the right of the text box.
This creates a new method and links it to the event. The Method Editor opens automatically (or it becomes active if it's already open).

You can write a new method to link the current event in the Inspector, or you can display the Edit Event dialog box to link the event to an existing method.

Note A method is a function defined in a class. The Form and Report designers are object-oriented; forms and reports are classes. Therefore all methods are defined and appear in the Method Editor with the reserved word function, and are sometimes (loosely) referred to as "functions."

The Method menu

Related topics

The Method menu offers commands to simplify writing methods in the Method Editor. The Event commands offer a dialog box as an alternative to using the Inspector to edit a selected objects events. You can also display these menu options (along with cut, copy, paste, and Method Editor properties) in a context menu by right-clicking within the Method Editor.

New Method
Creates a JavaScript skeleton for a new method in the Method Editor:

```
function Method()
{
// {Export} This comment causes this function body to be sent to the client
}
```

The new method is initially named “Method”; the name should be changed to the name you want. The method also contains the “{Export}” comment, which causes the method to be exported as client-side JavaScript. If the method is a server-side method only, you should delete that line.

Remove Method
Deletes the selected method and all references to the method from the script.

Verify Method
Attempts to compile the method, to make sure there are no syntax errors.

Edit Event
Displays a dialog box that allows you to select objects in the left pane and, in the right pane, select one of the available events for editing. The selected event is then displayed in the Method Editor for editing.

Link Event
Displays a dialog box similar to the Edit Events dialog box. You choose a control from the left pane and one of its events in the right pane. When you click OK, the new event is listed next to the method name at the top of the Method Editor.

Unlink Events
Displays a dialog box that allows you to view multiple events linked to a method and to remove any or all of them.

When you click OK, the selected link disappears from the link window next to the method name at the top of the Method Editor.

Using multi-page forms

Related topics

It is easy to create forms with several pages and navigation buttons.

When you create a new form, the Form Designer opens it on the first page. To create a multi-page form, choose the Next Form Page button on the toolbar. The Form Designer appends a page each time you click the button.

To navigate between pages in the Form Designer, use any of these techniques:

- Use the Next Form Page and Previous Form Page toolbar buttons.
- Choose View|Previous Form Page or View|Next Form Page.
- Use the PgUp and PgDn keys.
- In the Inspector, select the form object in the top selection box, and on the Properties page, change the numeric value of the *pageno* property. Notice that as you change this value, the pages of the form change on the design surface.

Global page

Related topics

In a multi-page form, page 0 is a “global” page. To create a recurring element or motif that repeats on all pages of a multi-page form, place the motif components on page 0. Controls on page 0 are visible on every page of the form.

To open page 0,

1 Select the form object in the Inspector’s top selection box.

2 On the Properties page, change the numeric value of the *pageno* property to 0.

Page 0 displays a composite view of all controls from all pages to help you position global controls so they will not interfere on the other pages. If you have several pages, naturally the various components of those pages may overlap in this composite view. To reposition the controls on other pages, you must navigate to the appropriate page.

Note When you save a multi-page form, the page that is active becomes the default page at run time. **Therefore, make sure you return to page 1 before clicking Run.**

Navigation buttons

Related topics

You will probably want to provide a control, such as a button or graphic object, to enable users to navigate between form pages.

A simple solution is to create two standard HTML buttons at the top of the global page (*pageno*=0) of the multi-page form.

The easiest way to quickly generate buttons is to choose them from the dialog box when you use the Form Expert to create your basic, or foundation form. The Form Expert gives you a choice of standard HTML buttons or image-style buttons, preset to provide Next, Previous, First, and Last record navigation.

For details about JavaScript navigation properties, see Database access from forms

To create navigation buttons on a multi-page form,

1. Go to the global page, page 0 of the form. Select the form object in the Inspector's top selection box, and on the Properties page, change the numeric value of the *pageno* property to 0.
2. On the Component Palette, Standard Controls page, drag the button control to the visual design surface. Add a second button. Ensure that the buttons will not overlap controls appearing on subsequent pages.
3. Select the first button and view in the Inspector (form.button1).
4. Make sure the numeric value of this button's *pageno* property is set to 0 so that it will appear on all pages.
5. On the Properties page, select the *text* property and type "Prev Page" (this replaces the placeholder text "button1").
6. On the Events page, select the event *onServerClick* and click the tool button to display the Method Editor. Enter this method:

```
function prevButton_onServerClick()
{
  this.form.pageno--;
}
```

   Note that the code following "pageno" is two hyphens, that is a double-minus. This means that when the user clicks this button, the previous page will be displayed.

7. Select the second button and view in the Inspector (*form.button2*).
8. Make sure the numeric value of this button's *pageno* property is set to 0 so that it will appear on all pages.
9. On the Properties page, select the *text* property and type "Next Page" (this replaces the placeholder text "button2").
10. On the Events page, select the event *onServerClick* and click the tool button to display the Method Editor. Enter this method:

```
function nextButton_onServerClick()
{
  this.form.pageno++;
}
```

    Note that the code following "pageno" is two plus signs, that is a double-plus. This means that when the user clicks this button, the next page will be displayed.

11. Return to page 1 before running the form. Select the form object in the Inspector's top selection box, and on the Properties page, change the numeric value of the *pageno* property to 1.

File operations

Related topics

After you create your form and work with the controls, you'll want to perform some file operations. You'll want to save your form, change the form, abandon some changes you make, run the form, and then print it.

Modifying a form

Related topics

When you start using a form you've created, you inevitably find ways to improve on your design. As a result, you might want to add a new control, rearrange component positions, change form properties, change fonts or colors, and so on. Fortunately, it's easy to change everything on the form using the Form Designer. Follow these steps:

1. Open a form in Design mode.
2. Choose File|Open, select the form you want to change, select Design Form, and choose OK. The Form Designer displays the form as it's currently designed.
3. Change the form components you want.
4. When you finish, save your changes.
5. Run the form to test its operation.

Saving changes to a form

Related topics

Save the form design to keep the changes you've made. If you have not yet saved a new form design, saving the design creates a new form (JFM) file.

You can save the form design in either of these ways:

- Click the Save toolbar button.
- Choose File|Save As to save the form under a new name.

If you haven't named the form, enter a file name. Choose a destination drive and directory, if needed, then choose OK. The Form Designer creates or updates the form (JFM) file.

Abandoning changes

Related topics

Abandon changes to a form design if you want to cancel creating a new form or discard the changes you've made to an existing form. To abandon changes,

1. Choose File|Close.
2. Choose No when prompted to save any changes.

Running a form

Related topics

To use a form for entering data, you *run* it. If you are designing a form, you can run it to see the results of your changes and to test its operation. You can switch easily between Design and Run modes.

If you run a form you haven't saved yet, the Save Form dialog box appears so that you can save it first.

To open a form in Run mode, do one of the following:

- Choose File|Open, select the form you want to run, select Run Form, and choose OK.
- In the IntraBuilder Explorer, click the Forms tab and then double-click the form button. The form appears in Run mode.

Printing a form

Related topics

Print a form in Design or Run mode by doing one of the following:

- Click the Print toolbar button.
- Choose File|Print.

Customizing

Related topics

IntraBuilder is versatile. You can create custom form classes, called “base forms” that serve as templates (with standard elements, such as company logos, animated GIF files, links, and so on, preset and ready to go) for creating new forms with the same look-and-feel. You can also create any number of custom components for use in your tables, forms, and reports.

In addition, you can customize the IntraBuilder interface to work more the way you do, by setting the properties for various IntraBuilder tools and components.

Using custom form class to create base forms

Related topics

When you create a form in IntraBuilder, the Form Designer opens a new, empty form by default. The Form Designer uses the Form class as the base form for all new forms.

You can create a custom form and specify it as a new base form—a custom form class—for use in the Form Designer. For example, if you want many forms in your application to have a similar look and feel, you can specify all the common attributes for those forms, such as colors, size, controls, and so forth, once. When you have established all the common attributes, save that form for future use as a custom form class. Thereafter, any changes you make to the custom form class will be reflected in all its derived forms.

To create a new base form,

1 Use the Form Designer to create the common features of the form.

2 Choose File|Save as Custom to display the Save as Custom dialog box.

3 Choose Save Form as Custom, then complete the rest of the dialog box

Or, double-click an (untitled) item with the empty Custom Form Class icon from the Forms page of the IntraBuilder Explorer.

Or, drag the (untitled) Custom icon onto the IntraBuilder desktop.

This opens the Custom Form Class Designer, which is almost identical to the Form Designer, except that it creates base forms with the JCF extension.

Then add the common features you want to appear on all derived forms.

Note You cannot run a base form; a base form is simply a template from which other forms can be derived.

To use a new base form,

1 Choose File|Set Custom Form Class.

2 Complete the Set Custom Form Class dialog box and choose OK

The new custom form class will be saved with the JCF extension to distinguish it from the active form, JFM.

Custom components

Related topics

You can create your own customized components and add them to the Component Palette for easy reuse. A custom component is based on one of the control or data access objects available from the Component Palette, but with preset properties, event handlers, or methods that you wish to reuse. For example, a Query object with an elaborate bit of SQL selection criteria for a particular table could be saved and quickly placed in other forms, ready to go.

You create custom components by adding a component to the form design surface and setting its properties, events, and methods for the appearance and functionality you want for your reusable custom component. Then you save it into a custom component file (with the CC extension) and add it to the Component Palette for convenient access.

You can use either a single control or a group of controls as a custom control.

To create custom components

Related topics

1 Use the Form Designer to create or customize the component or group of components you want to save. Drag the components you wish to customize to the form design surface.

2 Set each component's properties. (For example, the Button object's *onServerClick* event linked to a method that performs a special task that you will want to reuse elsewhere.)

3 Select the component or group of components in the Form Designer.

4 Choose File|Save as Custom to display the Save as Custom dialog box, then complete the dialog box and choose OK

5 Type a class name for your customized component and an entire path name to the file (with the CC extension) in which you wish to store this component. Click OK.

The custom component is now stored in the CC file you specified.

To add custom components to the Component Palette

Related topics

You can place, change the properties of, and attach event-handling code to the events of custom controls just as you can with standard controls. By clicking the check box in the Save As Custom dialog box, you can display your new custom control in a new Custom page of the Component Palette.

To load a custom component (stored in a CC file) and have it appear on the Custom page of the Component Palette, you need to select the CC file you want to load. Here is the procedure for adding custom controls and data access objects to the Component Palette:

1. Right-click in the Component Palette to display the Component Palette context menu.
2. **Choose Set Up Custom Components. (You can also access this command from the File menu.)**
3. **In the Set Up Custom Components dialog box, click Add. The Choose Custom Component dialog box appears.**
4. In the Choose Custom Component dialog box, choose the custom component file (with the CC extension) that you created for (or into which you saved) your custom component. Click Open.
5. **The path name to the selected custom component file now appears in the Set Up Custom Components dialog box.**
6. **Click Add. The custom components you have saved in your CC file appear on a new Custom page of your Component Palette. Now, whenever you need their preset functionality, just drag one of your custom components to the form design surface.**

Form Designer properties

Related topics

You can customize the user interface of the Form Designer to suit your preferences. To set the Form Designer properties, do one of the following:

- Choose Properties|Form Designer Properties.
- Right-click in the Form Designer window and choose Form Designer Properties from the context menu

Form settings

Related topics

These settings set the characteristics of the grid and ruler. By default, all are checked.

- Show Grid. When checked, it displays the grid.
- Snap To Grid. When checked, the corners of the object snap to the nearest grid intersection point when you add, move, or resize an object.
- Show Ruler. When checked, it displays the rulers on the left and top of the design surface.

Grid settings

Related topics

Determines the size of the grid units. The settings are calculated relative to the current form font.

- Fine. One-third the height and average width of the current form font.
- Medium. Two-thirds the height and average width of the current form font.
- Coarse. Equivalent to the height and average width of the current form font.
- Custom. User-defined size, relative to the form font. You set the size in the X and Y Grid spin boxes.

X and Y grids

Related topics

X determines the width and Y determines the height of the grid units. You can adjust these very precisely in the Form Designer Properties dialog box.

Field Palette properties

Related topics

1. **Right-click the Field Palette.**
2. Choose Toolbars and Palettes from the context menu. The Toolbars and Palettes dialog box appears.
3. **In the left pane, “Toolbars and Palettes,” click Form Designer-Field Palette.**

You can set these three options:

- Text Only

Click this radio button to make the Form Designer Field Palette display only the text names of the current active fields.

- Image And Text

Click this radio button to make the Form Designer Field Palette display both the text names and the icons of the current active fields.

- Mouse Revert To Pointer

Click this check box to make the mouse revert to a pointer after you have dropped a component icon on the design surface. (You might want this off if you are creating many instances of each component object.)

Component Palette properties

Related topics

To customize Component Palette properties, right-click in the Component Palette and choose Toolbars and Palettes.

Text editor properties

Related topics

There are four tabs of controls to customize the functionality of all IntraBuilder text editors, including the Script Editor, Method Editor, Script Pad, and Text Editor. You can set Fonts and Appearance, and color-code various language elements for improved readability.

Note These property settings affect all four editors.

The Editor page of the Properties dialog box (same for all text editors) lets you precisely limit line length, tab size, block indent, and the number of undo operations. You can also set mouse speed for the selected text editor, along with many other settings. You can save custom settings groups for different editing purposes.

The following table explains code editing preferences for the text editors:

Editor settings

Editor settings	Explanation and usage
Editor Speed Setting	This drop-down listbox lets you switch base editors. Choices are IntraBuilder and BRIEF. There are minor differences—some keyboard shortcut mappings, for example—but in most major respects the two editors offer similar functionality.
Reset	Revert editor preferences to default settings.
Auto-indent	Automatically aligns indented lines of code. When you press Enter at the end of a tab- or space-indented line, the new line is automatically indented to the same level. To indent further, press Tab. To move the insertion point back one indent level, press Shift+Tab.
Backspace outdents	If checked, lets you skip over indenting by pressing the Backspace key.
Optimal fill	If checked, converts groups of spaces to tabs when you load a file. If unchecked, spaces are preserved.
Use Tab character	Toggles the use of tab and the equivalent number of spaces.
Cursor through tabs	Determines what happens when an arrow key is pressed at a tab mark. If checked, the arrow key take you through the tab one character at a time. If unchecked, the arrow keys move the insertion point across the whole tab. When you press the right or left arrow keys and you're at the beginning or end of a tab, this setting determines whether the arrow key moves you through the tab space by space or whether you skip to the end of the tab.
Smart tab	Determines whether the tab key positions the insertion point to the starting column position of the previous line when you press tab and you are to the left of that point.
BRIEF cursor shapes	Toggle between the BRIEF style (horizontal) and IntraBuilder (vertical) cursor shapes.
Group undo	Determines whether all preceding editor commands and actions of the same type that have been executed since the last time Enter was pressed are “undone” when you choose Edit|Undo. If Group Undo is false, then only the last keystroke or command is undone.
Keep undo after save	Normally, when a file is saved, the undo cache is cleared. That is, any edits you made before the save cannot be undone. This option lets you override that behavior, allowing you to undo your most recent actions even after saving a file.
Persistent blocks	If columnar mode is on and this option is checked, a highlighted block of text remains highlighted until you select a new block. If the option unchecked, or if columnar mode is off, a highlighted block is automatically deselected when you click an area outside of the block. As noted, the Persistent Blocks option is only available when columnar mode is on. To toggle columnar mode on or off, press Alt+C.
Overwrite blocks	Replaces a marked block of text with whatever is typed. If Persistent Blocks is also on, then typed text is added rather than substituted for the marked text.
Cursor beyond EOF	If checked, lets you place your cursor anywhere on the page. If unchecked,

	the cursor cannot be placed beyond the last entered line.
Use syntax highlighting	Determines whether syntax highlighting and formatting settings are applied to files with a DBF source file extension. Untitled files edited in the Method Editor and text typed into the Script Pad all assume syntax highlighting when this option is checked. Existing files with non-DBF source extensions do not use syntax highlighting.
Visible right margin.	Adds a vertical line in the editor window to mark the position of the right margin. To change the position of the marker, use the Right Margin spinbox Default is 80 points from the left.
Interpret Text As	Choose DOS or Windows text conventions.
Mouse Speed	Drag the pointer to increase or decrease the speed with which the pointer moves over text.
Line Length	Specify the maximum line length in the text and script editors as well as the Script Pad input pane. The setting is applied to new edit windows or a reopened Script Pad; it does not apply to the current window or any open editing windows). If you type beyond this line length or paste data into the window that contains any line that exceeds the maximum, an error message appears.
Tab Size	Specify the tab width. This setting is applied immediately to all edit windows, including the Script Pad input pane.
Block Indent	Specify the indent of code blocks.
Undo Limit	Specify maximum number of bytes in the temporary cache that contains all current data available for Undo operations in any edit window.

The dialog boxis identical to those for the Script Editor and Method Editor and offers most of the same options for the Script Pad.

To create your own color scheme for highlighting language elements, choose a language element from the left pane, then select text attributes (bold, italic, underline), and if you wish, a color that becomes the “foreground” color for the text of that element. Your resulting color scheme is previewed in the Sample window at the bottom of the dialog box.

Designing reports introduction

Related topics

Reports display summaries of table data generated by one or more queries. They can display data in categories and perform subtotals on grouped fields. Unlike forms, reports are static. They are for display and printing only and don’t have interactive components. Report output is rendered into HTML pages that can be viewed through a Web browser.

Designing a report is similar to designing a form; the design environment and tools used in both processes are similar.

The most efficient method for creating reports is to first run the expert to create the basic report, then make changes to the resulting report using the designer.

Using the Report Expert

Related topics

To use the Report Expert to create a new report,

Click the Reports tab of the IntraBuilder Explorer and double-click the (Untitled) report.

The New Report dialog box appears.

Click Expert. The Report Expert opens to Step 1, which asks you to select a table or query to use as the source for the data in the report.

Select a table

Related topics

In the Look In box, click the folder icon and select the directory that contains the table or query you want to use. Then in the lower box, highlight the table or query and click the Next button.

Select a report type

Related topics

Include Detail Rows lets you include detail rows in the report. You'll pick the groupings and summaries used in the report in subsequent steps of the expert.

Summary Only gives you summary information for each group in the report (in which case, no fields would be shown in the report, only the summary).

When you have made a selection, click the Next button.

Select fields

Related topics

Select which fields (columns of the query) that you want included in the report. The fields in the source table are displayed in the Available list.

To include all the available fields, click the double-arrow button; all the field names move over to the box on the right. To move one field at a time, double-click it, or select it and click the single-arrow button.

When you have selected fields, click the Next button.

Add Groups

Related topics

You can group the records on any field. If you want the report to be sorted on a specific field, click that field. If you want the report to display records in the same order as the source table, don't click any fields in the Available box.

When you are finished, click the Next button.

For more information refer to Using Groups.

Add Summaries

Related topics

This dialog box allows you to select the summary information for each group. You can display an aggregate operation, like the number or items in each group (the count) or the minimum value of a specific field (for example, the lowest employee number in the group from California).

For more information refer to Using Groups.

Choose a layout style

Related topics

This step allows you to select the layout style for the report. The choices are Tabular (default) and Columnar.

You can also change the report's title, add a date, add a page number, and select either continuous records on a page or one record per page.

Choose Run or Design

Related topics

This step allows you to either run the report or enter Design mode to make further changes.

You now have a usable report that includes the basic elements. The records from the table will be displayed. The report is ready to be printed or deployed over the Web so that users can view it on their browsers.

Using the Report Designer

Related topics

In the Report Designer, you can perform any of the operations that the expert can, plus several more. The Report Designer tools are contained in the menu selections, the toolbar, the Field Palette and the Component Palette. Though there are some differences, the functions in the Report Designer are a subset of the functions in the Form Designer.

IntraBuilder's Report Design mode displays the report and the hierarchy of the groups on the report in separate panes. In Design mode, the Component Palette and the Field Palette are also available.

IntraBuilder displays the report in two ways:

- Group View
  The left pane is the Group View. It shows the hierarchy of all the groups in the report.
- Report View
  The right pane is the Report View. It shows the records displayed in a design block, optionally arranged in sections.
- Outer dotted line represents the margins of the actual report page.
- Inner dotted line represents the stream frame (the data records of the report).
- Bands represent the columns of the dataset that define the individual fields of the record.

You can move the dividing line (called the split bar) between the Report View and the Group View. When IntraBuilder starts, the split bar between the views appears all the way to the left side of the IntraBuilder window. Put the pointer directly on the left border of the IntraBuilder window; the pointer changes to a horizontal two-headed arrow. Click and drag the split bar to the right. In this way, you can make either of the panes bigger. You can move the split bar all the way to the left or right.

To view the two palettes, choose View|Component Palette and View|Field Palette.

Component Palette

Related topics

Use the controls on the Component Palette to place objects on the report. The Component Palette contains the same controls no matter which report you are working on. There are three tabs on the Component Palette: Standard, Data Objects, and Report Objects.

For details about components and data access objects, see Form Designer.

- **The Component Palette: Standard page**Pointer: Changes the cursor back to a pointer to select objects.
- CheckBox: Adds checkbox components to the report to display non-exclusive True/False data.
- Radio: Adds radio buttons to the report to display exclusive, dependent data selections.
- Rule: Adds horizontal rules (straight lines) as a divider on the report.
- Image: Adds a graphic frame for JPG or GIF images on the report.
- HTML: Adds an HTML label on the report for formatted titles, field labels, and HTML links.
- **Component Palette: Data Access page**Pointer: Changes the cursor back to a pointer to select objects.
- Query: Adds a query object on the report. The query object links a rowset from a table selected in the Inspector.
- StoredProc: Adds a stored procedure object on the report.
- Database: Selects a database.
- Session: Creates a session.
- **Component Palette: Report page**Pointer: Changes the cursor back to a pointer to select objects.
- Group: Adds a group to a report.

The Field Palette

Related topics

The components on the Field Palette place active fields on the report. Because the Field Palette's components each represent one of the fields in the source table, the Field Palette will look different when working on different reports.

The Field Palette has a page for each active query on the report. If there are no active query objects on the report, the Field Palette is empty. You must add a query object to link a table before the Field Palette can offer active fields linked to the table.

The easiest way to link a table to a report and populate the Field Palette with active fields, is to drag a table icon from the IntraBuilder Explore or Windows Explorer and drop it on the report. This automatically creates a Query object, set to active, with its *sql* property selecting all the table's fields.

Alternatively, you can double-click a Query object on the Component Palette or drag it to the report design surface. Then, in the Inspector, set the Query object to active and enter an SQL statement (or statements) in the *sql* property. This process, including the SQL Statement Builder, is described in Form Designer.

Drag a field from the Field Palette to the place on the report where you want that field to be, then release the mouse button. The new field will appear on the report as a new text field.

Report Designer menus and toolbar

Related topics

The menus and the toolbar change slightly while in the Report Designer.

The menu selections specific to the Report Designer are View, Layout, and Method.

Report Designer View menu

Related topics

View menu

- Report: Puts the report in Run Mode.
- Report Design: Puts the report in Design Mode.
- Inspector: Displays the Inspector (Properties, Events, and Methods).
- Method Editor: Displays the Method Editor.
- Zoom: Displays a submenu with Normal, Enlarged, and Reduced window settings.
- Component Palette: Displays/hides the Component Palette.
- Field Palette: Displays/hides the Field Palette.
- Toolbars: Displays the toolbar configuration dialog box.
- Status Bar: Displays/hides the status bar.
- IntraBuilder Explorer: Displays the IntraBuilder Explorer.
- Script Pad: Displays the Script Pad.

Report Designer Layout menu

Related topics

Layout menu

- Align displays a submenu that includes
- Left: Aligns the selected objects at their left edges (also available from the toolbar).
- Right: Aligns the selected objects at their right edges (also available from the toolbar).
- Top: Aligns the selected objects at their top edges (also available from the toolbar).
- Bottom: Aligns the selected objects at their bottom edges (also available from the toolbar).
- Absolute Horizontal Center: Aligns the selected objects in the horizontal center of the page.
- Relative Horizontal Center: Aligns the selected objects in the center of the page.
- Absolute Vertical Center: Aligns the selected objects in the vertical center of the page.
- Relative Vertical Center: Aligns the selected objects in the center of the page.
- Size displays a submenu that includes
- Grow To Largest Width
- ShrinkTo Smallest Width
- Grow To Largest Height
- Shrink To Smallest Height
- Add Groups and Summaries: Displays the Add Groups and Summaries dialog box that contains
- The Groups page: lets you add new groups to the report.
- The Summaries page: lets you add summary information to the report.

Report Designer Method menu

Related topics

Use the Method menu to edit and manipulate the methods linked to the objects on the report. The use of methods and the Method Editor is covered in Form Designer.

Method menu

- New Method
  Opens the Method Editor, which allows you to create a new custom method. Opening the Method Editor also enables the Remove Method and Verify Method options. At the top left of the Method Editor window is a list box that shows all the methods attached to the report.
- Remove Method
  Unavailable until a method is opened in the Method Editor. This selection deletes the method currently displayed in the Method Editor.
- Verify Method
  Unavailable until a method is opened in the Method Editor. After the Method Editor window is opened, the Remove Method and Verify Method selections become available. This selection runs a syntax check on the method currently displayed in the Method Editor.
- Edit Event
  Opens the Edit Event dialog box, which presents all the objects on the report and their events. Select an object from the Object pane on the left and that object's events are listed in the Event pane on the right. Notice that some objects don't have any associated events. Select an object and one of its events, and the Method Editor will open with the selected event in its current form. You can edit the event to perform specific operations.
- Link Event
  Displays the Link Event dialog box, which establishes links between events and methods. If you want a given event to trigger a method, you can link that method to multiple events. The Link Event dialog box displays the name of the method currently displayed in the Method Editor, all the objects on the report in a pane on the left, and each object's events in a pane on the right. While editing a method in the Method Editor, select Link Event from the Method Menu, then select an object in the left pane, and an event in the right pane, and the method will be linked to that event.
- Unlink Events
  Displays the Unlink Events dialog box which is used to break a link between a specific method and specified events. The Unlink Events dialog box displays the method currently in the Method Editor at the top of the box and the events linked to that method in the pane below. Select the events that you want to dissassociate from the method and click the OK button.

Complete information on editing methods and events is available in the online Help.

Using groups

Related topics

Grouping records by a specific field is very convenient and is one of the things that makes viewing a report more informative than directly viewing a table. You can apply groups to the records on a report. For example, you could display the total sales of a product by state, or break out sales totals by specific customers.

Summaries work together with groups to present information about a group as a whole. A summary can present a cumulative total for a group (the total payroll for the Sales group, for example), or the maximum or minimum value in a group (the highest or lowest salary in Sales, for example).

Designing queries introduction

Related topics

In IntraBuilder, you can build SQL queries visually, step-by-step, using the powerful Visual Query Builder (VQB). The advantage of building queries visually is that you can build and execute complex queries without knowledge of SQL. Even if you are an expert in SQL you are insulated from learning the differences in SQL syntax between the different database systems supported by VQB.

With VQB, you can build queries incrementally. This means that you can start with a simple query, execute it, see the results and refine it. You can repeat this process until you get the query you want. This process, known as “drilling down,” is a common way of working with databases.

You can also use VQB as a tutorial for learning SQL. Because you can generate and display the SQL statements for the queries you create visually, you can understand SQL a lot faster by working with VQB.

With VQB, you can

- Select one or more tables to be used in the query. See Adding tables to a query.
- Select columns to query by using simple drag-and-drop.
- Reorder result columns or delete selected columns by using drag-and-drop.
- Specify different types of join conditions between tables. See Join dialog box.
- Generate an expression to be included in the SELECT list. See Expression dialog box.
- Specify multiple selection criteria. See query.
- Specify grouping and group criteria. See Group conditions in a query.
- Specify how the results of a query can be sorted by one or more columns. See Sorting query results.
- View the SQL statement for the query. See SQL window.
- Execute the query and browse the result. See Result window.
- Save the query in a file.

Note In all examples, we assume that the tables and columns exist as described. The tables are in an Access database called BOOKSHOP.MDB, available for download from the Web, at

http://www.borland.com/techsupport/intrabuilder/w_faq.html

From the Data group on this Web page, click “How do I obtain the database for Chapter 6 of the Developer’s Guide?”

Toolbar

Related topics

You can use the Visual Query Builder's toolbar to select operations to be performed. The toolbar icons and their functions are shown below.

Icon	Function
New	Create a new query.
Open	Open an existing query. You will be prompted to choose the query file to open. The query files created by VQB have a .QRY extension.
Save	Save the query. This option saves the query under its original name. If no name was specified when the query was saved the first time, it is saved as UNTITLED.QRY.
Save As	Save the query with a new name. You will be prompted to choose a file name. The extension .QRY is supplied by default.
Options	Set query options. For more information, see Result window.
Tables	Select tables to query. For more information, see Adding tables to a query.
Expr	Build a result column as an expression. For more information, see Expression dialog box.
SQL	Show SQL statement. For more information, see SQL window.
Query	Run the query. For more information, see Result window.
Exit	Exit VQB. You will be prompted to save the current query if you made any changes.

Adding tables to a query

Related topics

To add a table to a query, choose the Add Table icon from the toolbar. The Add Table dialog box appears.

The Add Table dialog box lists the names of tables in the selected data source. You can add a table to the query either by double-clicking on the table name or by selecting the table name and then choosing the Add command button.

Table frames for each table included in the query are displayed in the VQB workspace.

Add Table dialog box

Related topics Example

Add tables to a query by using the Add Table dialog box. You can add tables when

- You start building a query
- You modify an existing query

The Add Table dialog box is automatically displayed when you start VQB. To display the Add Table dialog box at any time, choose the Add Table icon from the toolbar.

The Add Table dialog box lists the names of all tables in the current data source. If you want to see the system tables as well, check Include System Tables.

1. To add a table to the query, do one of the following:
   - Select the table name from the list of tables displayed and choose Add.
   - Double-click the table name.
2. After adding the required tables, choose Close.

   The tables selected for the query are now displayed in the VQB workspace.
3. To add a column from one of the selected tables to the query, do one of the following:
   - Select the column name from the list of column names displayed in the table frame. Drag the column and drop it on the query grid at the bottom of the screen.
   - Double-click on the column name.

   The selected column appears in the query grid.
4. You can create joins between the tables selected for the query. For more information, see Join dialog box.

Add a table example

Query:

List the title, ISBN, and publisher ID of each book.

To perform this query, you must first select the TITLES table (because that is where you know the information exists) and then select the Title column, ISBN column and Publisher column. With VQB, perform the following steps:

1 Make sure that the Add Table dialog box is displayed and double-click the TITLES table name.

Alternatively, select the table name from the list of tables and choose the Add command button. A table frame is displayed in the query workspace.

2 Choose Close.

We need to select only one table for this query. The TITLES table frame appears displayed in the VQB workspace with the column names in the table.

3 Double-click the Title column in the table frame.

This adds the column name to the query grid. The Table Name row for the column shows TITLES, signifying that the column belongs to the TITLES table. Option shows Show, signifying that the column's value will be displayed in the query result.

4 Select the ISBN column and drop it on the query grid. This is an alternative way to add a column to a query. When you drag the column name across the query workspace, the mouse pointer changes to  , signifying that the column cannot be dropped anywhere in the workspace. When you reach the query grid, the mouse pointer changes to  , signifying that it can be dropped in that position.

If you drop the new column on a column that is already on the query grid, the new column name is inserted before the existing column. If you place the new column in the blank area to the right of the query grid, it is appended as the last column of the query.

To select all columns from a table, double-click the table caption. The column names will be highlighted. Hold the left mouse button down and drag the column names to the query grid.

5 Add the Publisher column to the query.

6 Choose Run from the toolbar.

A result window is displayed with all records in the TITLES table with the field order you specified. You can scroll through the result set by using the vertical scroll bar. When you are finished looking at the result set, double click the system menu to close the result window and return to the VQB main window.

If you want to remove duplicate records from the result, select Option from the toolbar. In the Options dialog box displayed, check the Remove Duplicate Records check box and choose OK.

7 Choose SQL from the toolbar.

Note that the SELECT statement contains the three columns we selected. The SELECT statement in this case is very simple. It contains the SELECT keyword, a list of column names and the FROM keyword followed by the name of the table. As you learn how to use VQB, you will see more complex SELECT statements.

The SQL Statement window is a non-modal window. You can, therefore, return to VQB without closing the window. You can size the window suitably and position it on the desktop so that the SQL statement is visible always. If you change the query or selection criteria, the SQL statement will be updated immediately. If you are new to SQL, this is a good way to learn SQL syntax.

8 Choose Save As from the toolbar. In the Save Query dialog box, navigate to a directory where you want to save the query and save it with the name titles. The .QRY extension is automatically added by VQB.

For more information, see Adding tables to a query, Table names in a query, and Join dialog box.

Table names in a query

Related topics

The Table Name row in the VQB grid displays the name of the base table from which the query column is derived. If you have multiple tables included in a query, the table name corresponding to each column is displayed in the Table Name row. The column name itself is displayed right on top.

If you create an Expression column, the actual expression is displayed in the Table Name row. Each base column included in the expression is qualified by the table name in this case.

For more information, see Expression dialog box and Join dialog box.

Selection criteria for a query

Related topics Example

Specify the selection criteria for a query in the Criteria row of the query grid.

All selection criteria allowed in the WHERE clause of a SQL SELECT statement are valid. This includes =, >, <, !=, LIKE, BETWEEN, and IN clauses.

To make query selections that have to be AND-ed together, you specify multiple conditions in the Criteria row. To specify selection criteria that have to be OR-ed, you specify the selection conditions in the Criteria row and Or row.

Note You can enter up to 255 characters in a cell within the Criteria row. This means that the selection criteria specified for a column should not exceed 255 characters.

Examples

Example 1

Example 2

Selection criteria Example 1

Query:

List the title and ISBN of books whose titles include the pattern SQL.

1. Select the TITLES table by using the Table command.
2. Select the Title and ISBN columns to query.
3. In the query grid, bring the Criteria row into view using the vertical scroll bars.
4. Position the cursor in the Title column in the Criteria row.
5. Type LIKE '%SQL%' in the cell. Include the single quotation marks.

   This selection specifies that books whose titles include the pattern SQL anywhere in the title should be listed.
6. Choose Run.

   The result window displays the books whose titles contain the pattern SQL.
7. The SELECT statement for this query will be:

```
SELECT          Title , ISBN
FROM     TITLES
WHERE           ( Title LIKE '%SQL%' )
```

8. Save the query with the name sqlbooks.

Selection criteria Example 2

Query:

List the title, ISBN, publisher, price, and discount on books published by ADWE or that cost more than $35.

1. Select the TITLES table by using the Table command.
2. Select the Title, ISBN, and Publisher columns to query.
3. In the query grid, bring the Criteria and Or rows into view by using the vertical scroll bars.
4. Position the cursor in the Publisher column in the Criteria row.
5. Type 'ADWE' in the cell. Include the single quotation marks.
6. Add the Price and Discount columns to the query.
7. In the Or row, type >35 in the Price column.

   The selection criteria specify that books published by publisher ADWE or books costing more than $35 should be listed.
8. Run the query.
9. The SQL statement for this query reads:

```
SELECT   Title , ISBN , Publisher , Price , Discount
FROM     TITLES
WHERE    ( Publisher = 'ADWE' ) OR ( Price >35 )
```

10. Save the query as pubprice.

Query options

Related topics

The Option row in the Visual Query Builder grid is used for the following:

- To hide or show a query column. See Hiding a query column.
- To specify column aggregates such as COUNT, SUM, MAX, MIN, and AVG. See Specifying aggregates.
- To specify grouping criteria. See Grouping and aggregates example.

Hiding a query column

Related topics Example

While creating queries, you may want to select rows based on some selection criteria, but not want to display the value of the selection criteria in the query result.

For example, suppose you want to display the last names and ids of all employees in the production department. You would need the last name, id, and department columns to frame this query. However, you would not need to display the department column, because all records in the query result will have the same value for that column.

To hide a query column,

1. In the query grid, place the mouse pointer on the intersection of the column you want to hide and the Option row.
2. Click the mouse button.

   A pop-up menu appears with the Show menu item checked.
3. Click the mouse button to uncheck the menu item.

   The Show keyword is no longer displayed in the Option row. The selected column is not included in the query result. However, because the column is included in the query, you can still specify selection criteria.
4. The Show option works as a toggle. To re-display a hidden column, therefore, follow the above steps as they are. The Show pop-up menu item is now checked and the Show keyword is now displayed in the Option row.

Hide a query column example

Query:

List the title, ISBN and price of books on sale.

This query is based on the TITLES table. The OnSale column in the table indicates whether a book is on sale. If the column contains the value 'T', the book is on sale; if it contains the value 'F', it is not. The OnSale column has Character data type and therefore, the value entered should be enclosed in single quotation marks (').

The above query should, therefore, be phrased like this:

Find the Title, ISBN and Price for Titles where OnSale = 'T'.

Described below are the steps for building this query:

1. Select the TITLES table by using the Table command.
2. Select the Title, ISBN, Price, and OnSale columns to query. Use the vertical scroll bars on the table frame to bring the Price and OnSale columns into view.
3. In the query grid, bring the Criteria row into view by using the vertical scroll bars. If necessary, resize the VQB main window. You will not need the Group Condition or the associated Or criteria in this query.

   The Table Name, Option and Sort rows remain in view when you scroll vertically. These rows display information that you may want to refer to always.
4. Position the cursor in the Criteria row and the OnSale column. You can change the active cell either by using the mouse or by using the arrow/Tab keys.
5. Type 'T' in the cell. Include the single quotation marks.
6. Choose Run.

   The result window displays only those titles that are on sale.
7. View the SELECT statement for the query. It includes the row selection criteria.

```
SELECT   Title , ISBN , Price , OnSale
FROM     TITLES
WHERE    ( OnSale = 'T' )
```

8. Save the query with the name onsale.

   All rows in this query result contain the value 'T' in the OnSale column. You need not, therefore, display that column. To remove the column from the SELECT list:
9. Position the cursor in the OnSale column in the Option row.
10. Click the mouse button.

    A pop-up menu appears with the Show menu item checked.
11. Click the mouse button to uncheck the menu item.

    The Show keyword is no longer displayed in the Option row.
12. Run the query.

    The OnSale column does not appear in the query result.
13. The SQL statement for the query now reads:

```
SELECT   Title , ISBN , Price
FROM     TITLES
WHERE    ( OnSale = 'T' )
```

14. Using the Save command, save the query under its original name.

Specifying aggregates

Related topics Example

Group column values by specifying a GROUP BY condition.

To group column values,

1. Position the cursor in the column to group in the Option row.
2. Click the mouse button. A pop-up menu appears.
3. From the pop-up menu, choose Group.

VQB automatically assigns a group number for each column. If you remove a column from the query on which a group condition is defined, the GROUP BY condition on the column will be automatically deleted and the group numbers on other group columns automatically reassigned.

You can specify aggregate operations for non-grouped columns. These include SUM, COUNT, AVG, MIN and MAX**.** Aggregate operations are also selected from the pop-up menu.

To define an aggregate of a numeric expression, you can do one of the following:

- Create the expression using the Expression dialog box and then define the aggregate in the query grid.
- Define the expression and the aggregate directly in the Expression dialog box.

Grouping and aggregates example

Query:

List the number of orders and total value of orders for each customer.

To answer this query, you need to join the CUSTOMER, ORDERS and DETAILS tables. To get the number of orders for each customer, use the COUNT aggregate function. To get the total value of orders for each customer, use the SUM aggregate function.

To build and test the query,

1. Choose New to create a new query and choose the Add Table icon from the toolbar.

   The Add Table dialog box appears.

2. Add the CUSTOMER, ORDERS and DETAILS tables to the query and close the Add Table dialog box. For more information on how to add tables to a query, refer to the topic Add Table dialog box.

3. Define a join between the CUSTOMER and ORDERS tables using the CustomerId column. For more information on how to define joins, refer to the topic Join dialog box.

4. Define a join between the ORDERS and DETAILS tables by using the OrderId column.

5. Add the CustomerId, FirstName, and LastName columns from the CUSTOMER table to the query grid. Also add the OrderId column from the ORDERS table. For more information on how to add table columns to a query, refer to the topic Add Table dialog box.

6. Position the cursor in the CustomerId column in the Option row. Click the mouse button. From the pop-up menu displayed, choose Group.

   This step signifies that the query results will be grouped by the CustomerId column. The Option row shows Grp(1), indicating that the CustomerId column is the first GROUP BY column. If you have additional GROUP BY columns, they will be marked as Grp(2), Grp(3), ..., and so on.

7. In a similar way, define GROUP BY criteria for the FirstName and LastName columns.

   Remember that SQL syntax requires a GROUP BY condition to include all columns not involved in an aggregate operation. In this example, you need to define three GROUP BY columns.

8. Position the cursor in the OrderId column in the Option row. Click the mouse button. From the pop-up menu displayed, choose Count.

   This step signifies that the number of orders should be counted for each group.

   To get the total value of orders for each customer, you need to define an expression. To do so, proceed:

9. Choose Expr from the toolbar.

   The Expression dialog box appears.

10. Change Expression Name to *Total_Order_Value*.

11. Double-click sum(X) from the Functions list.

    The Expression edit box displays sum(X) with X highlighted.

12. Select DETAILS from the Tables pull-down list.

    The columns in the DETAILS table appear in the Columns list.

13. Double-click the column name Quantity.

    The column name replaces X in the Expression edit box.

14. Double-click the multiplication operator (*) in the Operators list.

    The multiplication operator is appended to the sum expression.

15. Double-click the SalePrice column in the Columns list.

    The column name is appended to the sum expression.

16 Choose Done.

You are prompted to save the expression definition. Choose Yes.

You return to the VQB main window. The Total_Order_Value expression is included as the last column in the query.

17 Run the query.

The number of orders and total order value for each customer appears.

18 The SQL statement for this query reads:

```
SELECT   CUSTOMER.CustomerId , FirstName ,
       LastName , count( ORDERS.OrderId ) ,
       ( sum( DETAILS.Quantity * DETAILS.SalePrice ) ) as Total_Order_Value
FROM     CUSTOMER , DETAILS , ORDERS
WHERE    ( CUSTOMER.CustomerId = ORDERS.CustomerId )
       AND
       ( ORDERS.OrderId = DETAILS.OrderId )
GROUP BY CUSTOMER.CustomerId , FirstName , LastName
```

19 Save the query as totalord.

For more information, see Expression dialog box.

Group conditions in a query

Related topics Example

In a query with GROUP BY conditions, you can perform selections on the aggregate columns.

For example, from an order database, you may want to create a list of salespersons whose total order bookings for the year exceed $1,000,000. This requires grouping the orders by salesperson and then selecting the salesperson records where the sum exceeds the specified target. To do so, you need to define a SUM aggregate for order value and then apply a selection criterion to the aggregate. This is the same as specifying a HAVING condition in a SQL SELECT statement with a GROUP BY clause.

VQB allows you to specify a HAVING condition on an aggregate column by typing in the selection criteria in the Group Condition row. The HAVING keyword is automatically added and need not, therefore, be specified.

Group conditions (HAVING clause) example

Query:

List the number of orders and total order value for customers whose total orders exceed $500.

To answer this query, we need to add a HAVING condition to the query demonstrated in “Grouping and aggregates example” on page 6-11. To do this, follow these steps:

1. Make sure that you have the totalord query in the VQB workspace.
2. Position the cursor in the Total_Order_Value column in the Group Condition row.
3. Type >500.

   This signifies that only those rows where the SUM of order value exceeds 500 should be listed.
4. Run the query.

   You will see the list of customers whose total orders exceed $500.
5. The SQL statement for this query reads:

```
SELECT   CUSTOMER.CustomerId , FirstName ,
       LastName , count( ORDERS.OrderId ) ,
       ( sum( DETAILS.Quantity * DETAILS.SalePrice ) ) as Total_Order_Value
FROM     CUSTOMER , DETAILS , ORDERS
WHERE    ( CUSTOMER.CustomerId = ORDERS.CustomerId )
       AND
       ( ORDERS.OrderId = DETAILS.OrderId )
GROUP BY
       CUSTOMER.CustomerId , FirstName , LastName
HAVING   ( ( sum( DETAILS.Quantity * DETAILS.SalePrice ) ) >500 )
```

6. Save the query as ordgt500.

For more information, see Grouping and aggregates example

Sorting query results

Related topics Example

You can sort query results in ascending or descending order of selected columns.

To sort query results,

1. Position the cursor under the column name on the Sort row.
2. Click the mouse button to get a pop-up menu.
3. Select Ascending or Descending from the pop-up menu. You can mix the Ascending and Descending options in one query.

You can specify up to 8 sort columns in a query. The number of columns for sorting may, however, be dependent on the database management system and the ODBC driver used.

Sorting example

Query:

List customer names, number of orders and total order value for each customer in descending order of total order value.

Note Some ODBC drivers and/or databases do not allow ordering on calculated column values. This query will not work with those databases.

To answer this query, you need to make use of an ORDER BY condition. You do this using the query demonstrated in Grouping and aggregates example.

1. Make sure that you have the totalord query in the VQB workspace.
2. Position the cursor in the Total_Order_Value column in the Sort row.
3. Click the mouse button and select Descending from the pop-up menu displayed.

   This signifies that the result should be arranged in descending order of total order value.
4. Run the query.

   You will see the list of customers, number of orders and total order value in the reverse order of total order value.
5. The SQL statement for this query reads:

```
SELECT   CUSTOMER.CustomerId , FirstName , LastName , count( ORDERS.OrderId
) ,
        ( sum( DETAILS.Quantity * DETAILS.SalePrice ) ) as Total_Order_Value
FROM     CUSTOMER , DETAILS , ORDERS
WHERE    ( CUSTOMER.CustomerId = ORDERS.CustomerId )
       AND
       ( ORDERS.OrderId = DETAILS.OrderId )
GROUP BY CUSTOMER.CustomerId , FirstName , LastName
ORDER BY 5 desc
```

6. Save the query as orddesc.

Join dialog box

Related topics Example

In many cases, you have to combine information from more than one table to perform a query.

For example, you may want to list all employees with their last name, employee id, and department name. However, the department name may be stored in the department table instead of the employee table. To perform queries of this type, you need to create *table joins*. VQB allows you to create joins using a simple drag-and-drop interface.

You begin a join operation by dragging the column name you want to join from the first table frame and dropping it on the target column name on the second table frame. When you drag the column name out of the first table frame and into the query workspace, the mouse pointer changes, signifying that the column cannot be dropped inside the workspace. When you reach the target table frame, the mouse pointer changes, signifying that it can be dropped in that position. When you complete the join, a line is drawn in the query workspace linking the columns in the two table frames. If you move the table frames in the query workspace, the line is automatically redrawn to indicate the join condition.

You can review and edit the join criteria by double-clicking on the line indicating the join. By default, the join is an inner join. If you want to change the join type to an outer join, double-click on the join line. The Join dialog box is displayed.

The Table1 and Column1 values identify the first table/column in the join. The Table2 and Column2 values identify the second table/column in the join. In case of an inner join, the order of table columns is not important, but it is important in case of outer joins.

Join Operator shows =. This specifies that the join is based on the equality of column values. You can specify a different join operator by selecting the corresponding option. For more information on various join operators, refer to the documentation of your database management system.

Join Type defaults to Inner Join. You can change it to other types of join supported by the database manager.

Note Some ODBC drivers support only one type of join known as inner join. Others support outer joins, but only one type of outer join known as left outer join. The Join Type options in the Join dialog box displayed by VQB reflect the capability of the driver to support different types of join. In case of dBASE, for example, only the Inner, Left Outer, and Right Outer options are enabled.

A table join can specify more than two tables, or a join between two columns in the same table. A join in which columns from the same table are referred to is known as a *self-join*. In case of a self-join, VQB adds a prefix such as __1, __2, and so on, to identify multiple instances of the same table. For example, if you create a listing of employees and their managers by using the EmpId and MgrId columns in the Employee table, the columns will be identified as Employee.EmpId and Employee__1.MgrId.

Instead of specifying an inner join (the default), you can perform an outer join between the tables in a query. To create the outer join, you need to edit the join definition in the Join dialog box. When you create a join by connecting two table frames, an inner join is created as default. To change the join type to outer join, double-click the line joining the table frames in the query workspace (or select the line by mouse-click and press Enter). The Join dialog box will be displayed.

To define the outer join, select the appropriate option button for Join Type. Only the outer join options supported by the ODBC driver are enabled and available for selection. After selecting the join type, choose OK to define the join. If you do not want to make a change, choose Cancel.

Note Because some ODBC drivers restrict the number of outer joins in a SELECT statement to one, you can define only one outer join condition in a query generated by VQB. If one outer join is defined, for all other joins in the query, the outer join option buttons will be disabled.

When you return to VQB after defining an outer join, the line that connects the table frames is shown in red.

Join example

Query:

List the subject, title, ISBN, author and publisher name of books on SQL.

In the sample database, the details of each book are maintained in the TITLES table. The information on the subjects covered by each book is maintained in the SUBJECTS table. The ISBN column is used as a link field between the two tables. To get the information required by this query, you would therefore need to specify that the SUBJECTS table be joined with the TITLES table by using the ISBN column.

To build and test this query,

1 Choose the New icon on the VQB toolbar to create a new query.

The Add Table dialog box appears.

2 Double-click the SUBJECTS table to add it to the query. Alternatively, select the table name from the list of tables and choose the Add command button.

The fields in the table are displayed in the SUBJECTS table frame in the workspace.

3 Add the TITLES table to the query as described in Step 2.

The fields in the table are displayed in the TITLES table frame in the workspace.

4 Close the Add Table dialog box by using the Close command button.

5 Arrange the table frames in the workspace so that you can easily work with them.

6 Select the ISBN column in the SUBJECTS table by clicking it. Then drag the column name across the workspace and drop it on the ISBN column in the TITLES table frame.

When you drag the column name out of the SUBJECTS table frame and into the query workspace, the mouse pointer changes, signifying that the column cannot be dropped inside the workspace. When you reach the TITLES table frame, the mouse pointer changes, signifying that it can be dropped in that position.

A line is drawn in the query workspace linking the ISBN columns in the two table frames. If you move the table frames in the query workspace, the line is automatically redrawn to indicate the join condition.

7 To view the join criteria, double-click the line joining the two columns.

The Join dialog box is displayed.

The Table1 and Column1 values identify the first table/column in the join. The Table2 and Column2 values identify the second table/column in the join. In case of an inner join, the order of table columns is not important, but it is important in case of outer joins.

Join Operator shows =. This specifies that the join is based on the equality of column values. You can specify a different join operator by selecting the corresponding option. For more information on various join operators, refer to the documentation of your database management system.

Join Type defaults to Inner Join. You can change it to other types of join supported by the database manager.

8 Choose OK in the Join dialog box.

9 Choose the Subject column from the SUBJECTS table to query.

10 Choose the Title, ISBN, Author and Publisher columns from the TITLES table to query.

11 In the Criteria row in the Subject column, type 'SQL'. Include the single quotation marks.

12 Run the query.

13 Here's the SQL statement for this query:

```
SELECT Subject , Title , TITLES.ISBN , Author , Publisher
FROM SUBJECTS , TITLES
WHERE ( SUBJECTS.ISBN = TITLES.ISBN )
```

```
  AND
( ( Subject = 'SQL' )       )
```

Note that the ISBN column in the SELECT statement's column list is qualified using the table name TITLES. Also note the use of the column qualifier for the ISBN column in the WHERE clause. This is required because the same column name is present in the SUBJECTS and TITLES tables. VQB automatically uses column qualifiers wherever required to identify column names unambiguously.

14 Save the query as titlesub.

Options dialog box

Related topics

The Options dialog box is used to

- Remove duplicate records from the result set. If you want to do this, check the Remove Duplicate Records check box. This is the equivalent of adding the DISTINCT keyword to a SQL SELECT statement.
- Delimit column and table names. This may be needed to work with databases that allow embedded spaces in column and table names, or to use SQL reserved words in column and table names. To enable the option, check the Always Quote Column and Table Names check box.
- Enable or disable the validation of table joins. To enable the option, check the Validate Joins check box. If this option is enabled, VQB verifies if the columns to join have compatible data types. If the columns have incompatible data types, you will receive an error message.
- Enable or disable the validation of selection criteria. To enable the option, check the Validate Criteria check box. If this option is enabled, VQB verifies that the selection criteria specified are syntactically correct for a SQL SELECT statement.

Expression dialog box

Related topics Example

You can define expressions as part of a query. These can be arithmetic expressions that perform calculations on numeric data values, or string expressions that concatenate strings or create substrings. Note that string expressions are supported differently by different database management systems.

To define an expression, choose the Expression icon from the toolbar to display the Expression dialog box.

Note You should select at least one table before building an expression.

VQB assigns a default name to each expression you define. You can change it by typing a different name in the Expression Name edit box.

The query can contain more than one expression. You select an expression to edit from the pull-down list.

The list of tables selected in the query will be shown in the Tables pull-down list. The columns of the table name shown in the Tables edit box are displayed in the Columns list. You can include any of the displayed columns as operands for the expression. To include columns from a different table, select the table from the Tables pull-down list and choose the required column from the Columns list.

In addition to column names, you can also include literal and numeric constants in the expression.

You can include the Addition (+), Subtraction (–), Multiplication (*) and Division (/) operators in an expression. You can change the precedence of these operators by including the operands in parentheses. To use any of the operators or parentheses in the expression, just double-click the item in the Operators list. It will be placed in the current cursor position in the Expression edit box.

The Expression edit box contains the definition of the expression. As you add more operands and operators, the Expression edit box will be automatically updated. You can directly edit the expression if you want to include literal or numeric constants.

You can also include the SQL aggregate functions in the expression: AVG, COUNT, MIN, MAX and SUM. Just select the function name from the Functions list. An X is placed as a place holder argument for the function. You have to replace it with the column name(s) on which the aggregate function is to be calculated.

In addition to SQL aggregate expressions, a number of built-in functions can also be included in the expression. Just pick the function name from the list displayed.

To save expression definitions and exit, choose Save. To exit without saving, choose Done. If you made changes, you are prompted to confirm an exit without save.

We create a simple expression in the example demonstrated for this topic. A more complex expression is demonstrated under the topic Specifying Grouping Criteria.

Expression example

Query:

List the item number, ISBN, quantity, selling price and extended price of the books in order number 10011.

In the sample database, the information on books ordered is maintained in the DETAILS table. For this query, this table alone is needed.

To build and test the query,

1 Choose the New icon from the toolbar to create a new query.

The Add Table dialog box appears.

2 Add the DETAILS table to the query workspace. Close the Add Table dialog box.

3 Add the OrderId, ItemNo, ISBN, Quantity and SalePrice columns to the query grid.

4 Choose the Expr command.

The Expression dialog box appears.

5 Change the Expression Name to Extended_Price.

Extended_Price is used as the name of the expression column in the query result.

6 Double-click the Quantity column in the Columns list.

The column name is automatically added to the Expression edit box.

7 Double-click the multiplication operator (*) in the Operators list.

The operator is appended to the Quantity column in the Expression edit box.

8 Double-click the SalePrice column in the Columns list.

The column name is appended to the multiplication operator in the Expression edit box.

9 Run the query.

You will see the list of books in order number 10011, along with the extended price of each order item.

10 Here's the SQL statement for this query:

```
SELECT   ItemNo , ISBN , Quantity , SalePrice ,
       ( DETAILS.Quantity * DETAILS.SalePrice ) as Extended_Price
FROM     DETAILS
WHERE    ( OrderId = 10011 )
```

11 Save the query as extprice.

Note The queries you create using expressions might not be portable across all databases.

SQL window

Related topics

The SQL Window displays the SQL SELECT statement associated with the current query.

To display the SQL Window, choose the SQL icon from the toolbar. As you add or change query columns, selection criteria, grouping, or sorting criteria, the SQL Window is automatically updated. Viewing the SQL statement gives you immediate feedback about the query design and also helps learning the SQL syntax.

Result window

Related topics

Run the query generated by VQB by choosing the Run icon from the toolbar.

The query results are displayed in the Result Window. This helps you verify that query columns, selection criteria, grouping, and sorting criteria have been correctly specified for the query.

Creating a home page

Related topics

In IntraBuilder, a home page is a form that helps you organize and present your IntraBuilder applications. The home page provides a directory or central organizing page with links to other IntraBuilder forms for different databases.

On the Web, a home page is the main entrance to a site, the logical starting point for a particular group of topics or for a specific group of people. A conventional home page bears the company logo and might have a link to each of the company's products or divisions.

This section of the Help file describes IntraBuilder's Home Page Form Expert, which makes building a home page quick and easy. The IntraBuilder Home Page Form Expert takes you through four steps that build a home page with a title, logo, e-mail address, and links to other forms.

Building a Home Page

Related topics

To build a home page, select File|New|Home Page Form. The New Home Page Form dialog box appears.

Click the Expert button to display the first step of the Home Page Form Expert. Click the Designer button if you want to design a home page from scratch.

Step 1

Step 1 presents several pre-built fields that will appear on the home page you are building.

Type text in the field titled Company Name for the title of the home page.

Use the Company Logo field to place a graphic image on the home page. Below the Company Logo field is a set of radio buttons that are used to place the graphic image in one of four possible positions with respect to the Company Name.

At the right are two boxes that will display text on the home page. Type a company motto or slogan in the top one and an e-mail address in the lower one. The fonts used and other formatting of the text is handled later.

After typing in the text you want, click the Next button to go to the next step.

Step 2

Step 2 gives you the opportunity to place links on your home page to other pages in your site.

Click the browse button to find the directory of forms and reports that you want to link to your home page.

You can put any number of links on your home page. Files with extensions of JFM JRP appear in the left box. Select the files and then click the right arrow button to move individual forms and reports to the Selected Links box. Each of the selected file names will appear in the box on the right.

Click one of the Selected Links file names and that name will appear in the Description box below the Selected Links box. You can accept the file name as the link name. Or you can type a new name for the selected link and that text will be used as the text on the actual link as it appears on your home page. The correct file name will be used in the HTML HREF link code.

Identify all the links you want to make, change the descriptions of each link (if you want), and then click the Next button.

Step 3

HTML:HREF link code;color scheme, creating;Scheme listbox (Home Page Form)

Step 3 lets you specify a scheme, or font style and color, for the title and labels that will appear on your home page. Each of the text fields that you put on your home page can be formatted independently.

You can use any of several pre-built color and font schemes for your home page. Or, you can design your own. Any of the formatting you do here, you can change later in the Form Designer.

The Sample box shows an example of the colors and fonts in the different schemes. Select a scheme from the Scheme listbox to see it in the Sample box. When you have a scheme that you like, click the Next button.

Step 4

At this point, your new home page is complete, and you can either run it or go to the Form Designer to make further adjustments. Click Run Form to see what your form looks like. The Save Form dialog box appears. Name the form and save it. After saving it, the home page appears in Run mode.

If you want to make changes, select View|Form Design. This puts you in the Form Designer. See Form Designer for guidance in modifying forms.

Security introduction

Related topics

Beyond the enterprise firewall system, your next line of defense is the security features of your Web server application. It is recommended that you take full advantage of these features to prevent unauthorized access to your Web server.

Once users have gained access to your Web server, IntraBuilder provides additional built-in levels of security against unauthorized access to encrypted databases and tables. This table-level security depends upon data encryption.

Sensitive tables should always be encrypted by using the database vendor's administration software. IntraBuilder's pre-built password forms are automatically displayed on the browser whenever a user tries to access a form linked to an encrypted table or database. The user's response to the password form is passed to the encrypted table or database for verification before IntraBuilder will display the form. See your database vendor's documentation about security administration for SQL, ODBC, or non-standard systems.

The DBF and DB tables you create within IntraBuilder have built-in encryption. IntraBuilder provides direct database administration security access to set passwords for standard DB and DBF tables, as well as the extensive user-access and privilege-level security features of DBF tables.

Finally, IntraBuilder's robust JavaScript lets you create powerful custom login forms for controlling access at the application level. You can write your own login forms to give custom top-level protection to IntraBuilder applications, including all or some of their constituent forms, tables, and reports. Although you could also write custom login forms for table-level access, it isn't necessary thanks to IntraBuilder's pre-built password forms.

Security strategies

Related topics

IntraBuilder offers two general strategies to handle access to encrypted tables of any type: individual login and preset access. Some intranet security strategies could involve using preset forms as well as both automatic password forms and custom login forms, as explained in Preset access via Database and Session objects and in Custom security.

This chapter discusses the following approaches to establishing security:

- **Individual login via automatic password forms**

In this approach, each user is required to login every time he or she tries to access a form linked to an encrypted table or database. IntraBuilder automatically displays a pre-built password form for the appropriate table type, requiring the user to enter a password or other information required by the table. Users might get different access levels, depending on their user name and password, and depending on the security features supported by the table type. The user must submit the correct information (which is passed to the encrypted database system for verification) before they can be accessed by the IntraBuilder form. You can customize these prebuilt forms by using Form Designer.

- **Preset access via Session and Database objects**

Preset access involves hard-coding passwords or user names in IntraBuilder forms and reports. Preset access provides an automatic pre-determined level of access without login procedures for certain groups of users. It can be used in conjunction with individual login to provide easy read-only access for the public and login-protected access for authorized company personnel.

- Preset access for standard table types

Sessions objects provide unique connections between a user and a DBF or DB table. You can add methods to these objects to restrict access to certain features of a standard table, or to make the table read-only for certain login-levels.

- Preset access for SQL and other table types

Database objects link IntraBuilder forms to SQL databases or table sets. You can set the Database object's *loginString* property for particular user names or passwords, to limit users of a specific login-level to read but not write the data in a SQL database.

- **Table-level security for DBF tables**

IntraBuilder supports direct access to the extended security features of DBF tables, including administrator security, and up to 8 user access levels, and three-level privilege security for DBF tables and individual fields. If you intend to create tables within IntraBuilder, DBF tables offer the most extensive and versatile security features.

- **Table-level security for DB tables**

IntraBuilder provides direct access to master password security for each DB table. However, you must use Borland's Paradox or Database Desktop to set auxiliary passwords.

- **Custom security**

You can write your own JavaScript-based login forms to restrict access to applications or particular tables, or to provide user identification or restrict certain features to a limited login. This strategy is detailed in Customizing the application. in which a three-page login form is created for the sample Threaded Message Database application.

Individual login via automatic password forms

Related topics

IntraBuilder's pre-built automatic password forms are activated only by encrypted tables. Password protection alone is inadequate to protect a sensitive table unless the table is encrypted, because an intruder, having gained access to the site, could use another application to read the data from the hard disk.

Once an authorized user gains access to your intranet site by providing the correct password, he or she might be offered a restricted a choice of a variety of tables, with different access privileges, depending on the login level. IntraBuilder supports the full range of table- and row-level security features for DBF tables, so you can create up to 8 user access levels and 3 privilege levels, precisely controlling access by different classes of users to specific tables and even to specific rowsets in those tables.

To link any encrypted table to a form (and thereby enable automatic password protection) you need only create a Query object for that table on your form. To do this, simply drag the table icon of the encrypted table from IntraBuilder Explorer's Tables tab to your form surface in Form Designer. (At this time, while in Design mode, you will have to supply access information to the encrypted table.) This is all you need to do to ensure that IntraBuilder will activate the automatic password form. See Form Designer for guidance on adding Query objects to forms.

After your form is deployed and the user activates some event (a button click, for example) to access a restricted table, the Borland Database Engine (BDE) attempts to open that table. Because the table is encrypted, IntraBuilder automatically displays the appropriate prebuilt password form with fields for the input required by that table type. The type of security available varies according to table type.

The original IntraBuilder form is temporarily displaced and the user is presented with a password form. To gain access to the form (and its underlying encrypted table), the user must provide the particular security information required by that table or database.

By using the Form Designer you can modify these automatic password files, adding additional text entry fields or changing their size to accommodate the security requirements of the table or database.

IntraBuilder includes three prebuilt password forms that may be automatically displayed on the user's browser when an encrypted table requires a login response:

- pass-dbf.jfm

A password form for DBF files, dBASE-type tables.
Corresponds to the sessions's *login*() method.

- pass-pdx.jfm

A password form for DB files, Paradox-type tables.
Corresponds to the session's *addPassword*() method.

- pass-sql.jfm

A password form for SQL and other table types.
Corresponds to the database's *loginString* property.

The automatic password files are located in:

```
C:\Program files\Borland\Intra\bin
```

IntraBuilder displays the password forms automatically only when attempting to activate a Database or Query object on the form. In other words, the automatic password forms do not appear if you activate the database or query in a method or an event handler.

You can customize the appearance of the automatic password forms in Form Designer, but you cannot change the information items requested. For example, you might use the same group name for everyone accessing a DBF table over the Web, but there is no way to preset the group name.

Preset access via Database and Session objects

Related topics

In addition to the individual login approach (where users must login whenever they run a form that accesses encrypted tables) you can setup various preset access levels for your Web applications.

This means hard-coding passwords or user names in IntraBuilder forms and reports to provide an automatic pre-determined level of access without login procedures for certain groups of users.

Preset access can be useful in combination with individual login (using either automatic password forms or custom login forms) to provide easy read-only access for the public and login-protected access for authorized personnel. For example, you might have employee information that is editable by your company's Human Resources staff through an application on the LAN, but this information would be restricted to read-only access over the Web by means of a preset form.

To implement this strategy you would need a full-access (read/write) password that the on-site personnel would have to enter manually every time they start the application. You would code into the form a read-only password that would admit everyone else at that limited level.

It is rather easy to code the preset access level. How you do it depends on the table type.

Preset access for standard table types

Related topics

For DBF and DB tables, security is session-based. The Session object has a *login*() method for DBF table security and a *addPassword*( ) method for DB table security. The appropriate method (or both if you're using both types of encrypted tables) must be called with the correct user name and password before attempting to activate a query that accesses the table.

Where this must occur depends on whether all users are sharing the same session. If everyone accessing the IntraBuilder Server gets exactly the same access for every application by using the same user name and password, then they can share the default session. You would need only call the session's methods once through an administrative script or form, after the IntraBuilder Server is started.

For example, the JavaScript statements would look like this:

```
_sys.databases[ 0 ].session.login( "Group", "User name", "Password" );  //
DBF
_sys.databases[ 0 ].session.addPassword( "Password" );                   // DB
```

All the forms would require a Query object to access the DBF or DB tables, but no Database object or Session object, because they're using the default database in the default session.

On the other hand, if any two applications use different user names or passwords then every form must have its own Session object, so that each form runs in its own session and the security is localized.

No Database object is needed because the form uses the default database of its own session. Then users must log into each session before the query is activated.

Use the Query object's *canOpen* event to call the session's security methods.

Preset access for SQL and other table types

Related topics

Table and database types other than DBF or DB tables accessed via the directory require modification of the Database object's loginString property. This applies to all non-standard application, including SQL servers such as Borland InterBase, Oracle, Sybase, Informix, IBM DB2, and MS SQL Server; and ODBC connections such as Access and Btrieve. It also applies to remote DBF and DB tables accessed through a Borland Database Engine (BDE) alias.

A BDE alias always identifies a database. Therefore all non-Standard table security is through the Database object that provides access to that database. In some cases, logins are required to access tables in a database.

The Database object's *loginString* property is a character string that contains the name and password in the form:

```
name/password
```

You can set this property in the Inspector in the Form Designer. By setting the name and password in the form's Database object, all users attempting to open that form will get whatever level of access that name and password provides.

Although possible, it's more trouble than it's worth to share a Database object among multiple forms. Each form should have its own Database object, with whatever the appropriate *loginString* is for that particular form.

Table-level security for DBF tables

Related topics

The security features of DBF tables are extensive. If you intend to create private tables within IntraBuilder for which you wish to set elaborate or varied access levels, the DBF table type is your best choice.

Table-level security relies on data encryption. Data encryption scrambles data so that it can't be read until it is unscrambled. An encrypted file contains data that has been translated from source data to another form that makes the content unreadable. If your database system is protected, IntraBuilder automatically encrypts and decrypts tables and their associated index and memo files when user supply the required passwords or other login information.

In addition, DBF tables allow you to define which fields within tables users can access, and the level of access, read, read/write, or full.

The first parts of this section describe how to plan your security scheme for DBF tables. Topics include:

- The various levels of security
- An overview of the various aspects of the DBF security
- Planning group access for each table
- Planning each user's login and user access level
- Planning user access to tables and fields within tables

At the end of this section are procedures for setting up your DBF security scheme:

- Enter the database administrator's password
- Create user profiles
- Set user privileges for table access
- Set user privileges for fields within tables

About groups and user access

Related topics

You can control access to individual DBF tables (and to fields within those tables) by carefully defining groups of users according to:

- Which tables each group can access
- Which privilege levels (read, update, extend, delete) each group has at the table-level
- Which fields within tables each group can access
- Which privilege levels (none, read-only, full) each group has at the field-level

Table access

Related topics

First, you'll need to define user groups and determine which group has access to which table. Try to organize users and tables into groups that reflect application use (for example, by department or sales area).

- A table can be assigned to only one group. If the user group and table group don't match, the user can't access the table.
- Typically, each group is associated with a set of tables. By associating each application with its own group, you can use the group to control data access.
- A user can belong to more than one group. However, each group that a user belongs to must be logged-in separately.
- If a user needs to access tables from two different groups in the same session, the user must log out of one group, then log in to the second. A user may have separate logins into different groups in separate sessions to access files in different groups.

User profiles and user access levels

Related topics

You'll need to develop a user profile for each user in each group. As part of each profile, you'll assign to the user an *access level.* Each user's access level is matched with the table's privilege scheme (see the next section) to determine what access the user has to the table and, within each table, to each field. For example, if you establish a read privilege of 5 for a table, users with a level from 1 to 5 can read that table. Users with a level of 6 or higher can't read the table.

By establishing access levels within a group, you can give different users different kinds of access to the table and to fields within the table.

- Access levels can range from 1 to 8 (the default is 1). Low numbers give the user greater access; high numbers limit the user's access. The access value is a relative one—it has no intrinsic meaning.
- The less restrictive levels (1, 2, 3) are typically assigned to the fewest people. To limit access to data, the more privileges a level has, the fewer users you should assign to that level.
- You can assign any number of users to each access level.
- If you don't need to vary the access level of the users within a group, there is no need to change each user's default level.

About privilege schemes

Related topics

Once you've established each users's access level, you set up a *privilege scheme* for each table. A DBF table's privilege scheme controls three things:

- Which group can access the table. (The user's group name is matched with the table's group name to allow table access.)
- Which user access levels can read, update, extend and/or delete the table (table privileges).
- Which user access levels can modify and/or view each field within the table (field privileges).

After a user logs in, IntraBuilder determines what access the user has to that DBF table and its fields by matching the user's access level with the rights you specified in the table's privilege scheme.

For example, if you assigned a user an access level of 2, that user's access to the table, and to various fields within the table, are determined by the privileges you assigned to Level 2 in the table privilege scheme.

In building a table privilege scheme, note that:

- A user's ability to access a table is a function of both the access level of the group and the user's individual access level. However, only the user's access level determines what the user can do with a table once it is opened.
- If you do not create a privilege scheme for a table, all users of the group can read and write to all fields in the table.
- Access rights cannot override a read-only attribute established for the table at the operating system level.

Table privileges

Related topics

At the table level, you can control which operations each user access level (1–8) can do:

- View records in a table (read privilege)
- Change table record contents (update privilege)
- Append new records to a table (extend privilege)
- Delete records from a table (delete privilege)

When you create a table privilege scheme, all four table privileges are granted initially. That is, all table access levels are 1 by default (1 being the *least* restrictive level).

Field privileges

Related topics

At the field level, you can control which operations each user access level (1–8) can do:

- Read and write to the field in the table (FULL privilege). This is the initial default.
- Read but not write to the field (READ ONLY privilege).
- Neither read nor write the field (NONE privilege). NONE blocks a user from writing to fields and from seeing fields you do not want to display.

About data encryption

Related topics

A DBF table is not encrypted until you select it, edit the access levels, and save the privilege scheme.

When a DBF table's privilege scheme is saved, IntraBuilder encrypts the table, including the production index (MDX) file and the memo (DBT) file, if any. IntraBuilder also creates a backup copy of the original, unencrypted table. To ensure proper security, the backup files should be archived, then deleted from the system.

Even after a database system has been protected, the database administrator and application programmer maintain control over encryption of copied files.

Planning your security system

Related topics

This section describes how to plan out your security system for DBF table security. It's a good idea to think through user access and table/field rights before you start creating security profiles.

Follow these general steps to set up a protected database system for DBF tables:

1. Plan your user groups.
2. Plan each user's access level.
3. Plan each table's privilege scheme, including both table privileges and field privileges.
4. Implement your security scheme (see Setting up your DBF table security system**).**

Planning user groups

Related topics

Take time to think through the various grouping into which you can divide your users, based on who needs access to which tables. For example, an administrative staff might need to access tables that a sales staff does not, or vice versa. Other groups may overlap; for example a marketing group might need to see some of the administrative tables and some of the sales tables.

It helps to develop a worksheet, to map group access needs in advance. The following table shows one way of organizing this information; use whatever method works best for you.

Worksheet for defining groups and group members

Table	Group	User name
CUSTOMER	SALES	AMORRIS
		BBISSING
		LJACUS
		FFINE
PRODUCT	ALL	AMORRIS
		BANDERS
		BBISSING
		CDORFFI
		LJACUS

Planning user access levels

Related topics

Next, think about how much access each user needs to the table.

Although there are 8 access levels, you might choose to standardize on just 3 levels; one for full access, one for typical use, and one for minimal access. The next table shows the sample worksheet, expanded to show user access levels.

Worksheet expanded to show user access levels to DBF tables

Table	Group	User name	Level 1 (full access)	Level 4 (typical access)	Level 8 (minimal access)
CUSTOMER	SALES	AMORRIS	X		
		BBISSING		X	
		LJACUS	X		
		FFINE			X
PRODUCT	ALL	AMORRIS	X		
		BANDERS		X	
		BBISSING		X	
		CDORFFI		X	
		LJACUS	X		
		FFINE			X

Planning DBF table privileges

Related topics

Next, plan each DBF table's privilege scheme.

For each table operation, determine the most restricted access level that can perform the operation. All levels less restricted than the specified one can perform that operation; all levels more restricted than the specified level cannot.

The following worksheet illustrates one way to plan which user access levels grant which table rights.

Worksheet for defining privileges for DBF table operations

Table	Read	Update	Extend	Delete
CUSTOMER	8	4	4	1
PRODUCT	8	4	4	1
ORDERS	8	4	4	1

Planning field privileges

Related topics

The last planning step is to determine which user access levels can read and/or write to fields. Consider developing a worksheet similar to the following one.

Worksheet for defining field access privileges to DBF tables

Field name	Full access	Read only	No access
PAYRATE	Levels 1–2	Levels 3–6	Levels 7–8
FIRSTNAME	Levels 1–6	Levels 7–8	
LASTNAME	Levels 1–6	Levels 7–8	
SSN	Levels 1–2	Levels 3–6	Levels 7–8

Setting up your DBF table security system

Related topics

Once you've planned out your security scheme for DBF tables, you're ready to set it all up. Follow these steps to implement the security scheme:

1. In IntraBuilder define the database administrator password.
2. Define the user profiles, including group membership and access level.
3. Define table privileges.
4. Define field privileges.
5. Set the login security scheme.
6. Save the security information.

This section describes how to set the database administrator password, how to enter and edit user profiles, and how to set up table privilege schemes.

Defining the database administrator password

Related topics

Before setting passwords, make sure any open tables have been closed. Follow these steps to enter the database administrator password:

1. Choose File|Database Administration. The Database Administration dialog box appears.
2. In the Database Administration dialog box, make sure that the Current Database field is set to <None> and the Table Type field is set for dBASE (DBF) tables.
3. Click the Security button. The Administrator Password dialog box appears.
4. In the Administrator Password dialog box, enter a password of up to 16 alphanumeric characters. You can enter characters in upper- or lowercase. The password does not appear onscreen.

   The first time you set the administrator password you are prompted to reenter the password to confirm. (Thereafter, the system gives you three chances to enter the password correctly before the login terminates.) The Security dialog box appears.

Warning! Once established, the security system can be changed only if the administrator password is supplied. Keep a hard copy of this password in a secure place. There is no way to retrieve this password from the system.

Creating user profiles

Related topics

The Security Administrator dialog box is where you create user profiles and establish an access level for each user.

Follow these steps to add a user profile:

1. In the Security dialog box, select the Users tab and click the New button.
2. Enter a user login name (1–8 alphanumeric characters) in the User field. The entry is converted to uppercase. Required.
3. Enter a group name (1–8 alphanumeric characters) in the Group field. The entry is converted to uppercase. Required.
4. Enter a password for this user (1–16 alphanumeric characters). Required.
5. Select an access level for this user (from 1 through 8; see About groups and user access**). Lower numbers give the greatest access; higher numbers are the most restricted.**
6. Enter the user's full name (1–24 alphanumeric characters). This entry is optional. Because this item is not used in validating a login, you can use it any way you want. Frequently, the full name is used to add a more complete user identification. Alphabetic characters you enter in the Full Name option are not converted to uppercase.
7. Click OK to save the user profile.
8. The Security dialog box reappears with your new user info added to the list in the Users tab.Repeat the preceding steps for each user.

Changing user profiles

Related topics

To change a user's profile,

1. Open the Users tab of the Security dialog box.
2. Select the user name of the user you want to change, and click the Edit button.
3. Make the desired changes, then click OK.

Warning! Be careful when editing the group name, deleting the group, or deleting all users from a group. If you edit the group name, there is no way for its users to access tables associated with the original group. And if you delete the group or all users from a group before all tables associated with the group are copied out in a decrypted form, no one can access the tables. In that case, you must create a new user for the group.

Deleting user profiles

Related topics

To delete a user profile,

1. Open the Users tab of the Security dialog box.
2. Select the user name of the user you want to delete, then click the Delete button.
3. To confirm the deletion, click the Yes button.

Establishing DBF table privileges

Related topics

Follow these steps to define table and field privileges for a table:

1. Open the Tables tab of the Security dialog box.
2. Select a table.
3. Assign the table to a group.
4. Establish the most restrictive access level for each table privilege.
5. Select field privileges for each user access level.

In general, for DBF tables you can use the Tables tab of the Security dialog box to:

- Assign a table to a specific group.
- Set table access privileges.
- Set field access privileges for each user access level.

The sections that follow describe these steps in detail.

Selecting a table

Related topics

To select a table:

1. Open the Tables tab of the Security dialog box. You use the Tables tab of the Security dialog box to create and modify DBF table privilege schemes. The DBF table privilege schemes are saved in the table structure.In the Table field, type the name of the desired table. (Or click the Tools button and select the table.)
2. Click the Edit Table button. The Edit Table Privileges dialog box opens.

Assigning the table to a group

Related topics

A DBF table can be assigned to only one group. The group name is matched with a user group name to enable data access.

To select a group for the DBF table, click on the down arrow to display a list of the available groups from the Group list in the dialog box. (These groups were created when you created user profiles.)

Setting DBF table privileges

Related topics

For each type of table operation (see the table below), specify the most restricted access level that can perform that operation.

Table operations

Privilege	Access granted
READ	View the table contents
UPDATE	Edit existing records in the table
EXTEND	Add records to the table
DELETE	Delete records from the table

To set table privileges, select a value (1–8) for each operation (Read, Update, Extend and Delete) in the dialog box. Remember that lower access numbers indicate the greatest access; higher numbers indicate the greatest restriction.

Note You cannot specify access levels that are logically incompatible. For example, you cannot prohibit Level 6 from having read access, but also permit Level 6 to have update access. To have update access, Level 6 also needs read access.

Setting field privileges

Related topics

With DBF tables you can establish access for each field by user access level. Table 8.6 describes the available field privileges.

Field privileges

Privilege	Access granted
FULL	View and modify the field. This is the default.
READ-ONLY	View the field only (no update capability).
NONE	No access. The user can neither read nor update the field, and the field does not appear.

Note Table privileges take precedence over field privileges. For example, if a table privilege is set for Read but not Update, the only meaningful field privileges are Read-Only or None. You must restrict *table* privileges to protect your data against table-oriented commands like DELETE and ZAP. Restricting field privileges to Read-Only or None without restricting table privileges doesn't protect data against these commands.

The Fields list in the dialog box lists all of the fields in the current table. The Rights buttons display the field privileges for the selected field for access levels 1 through 8. Initially, all field privileges are set to Full.

Follow this procedure to change a field privilege:

1. Select the field.
2. Click the Rights buttons that correspond to the privileges you want to grant for the field *for each access level*.

   For example, the rights as shown in Figure 8.5 set privileges for the field CREDIT_OK so that users with access Levels 1 and 2 have full access, users with access Levels 3 through 5 have read-only access, and users with access Level 6 and 8 have no access.
3. Repeat the process for each of the other fields in the table.
4. Click OK to save the field access privileges.

Warning! Never change the access rights of the _DBASELOCK field of any table. The rights to this field must remain Full for all access levels.

Setting the security enforcement scheme

Related topics

You can choose one of two enforcement schemes:

- When a user attempts to load IntraBuilder itself, a login is required, thus preventing unauthorized users from meddling in your intranet system.
- Whenever a user tries to view a form linked to an encrypted DBF table, a login is required. Thus anyone may use unencrypted tables, but unauthorized users are prevented from accessing protected tables.

To change the security enforcement scheme, follow these steps:

1. Open the Enforcement tab of the Security dialog box. The two radio buttons on the Enforcement tab indicate the security enforcement scheme currently in effect.
2. Select the enforcement scheme you want: whether to display a password form when loading IntraBuilder or only when accessing an encrypted table.
3. Click Close.

Table-level security for DB tables

Related topics

Although DB tables do not offer the extensive user access-level and privilege level security system available to DBF tables, DB tables (unlike DBF tables) support passwords.

You can use IntraBuilder to assign master passwords to DB (Paradox) tables. Once you have assigned a master password assigned to a DB table, it cannot be opened without supplying the password, either by you locally or by users over the Internet.

You may choose to create a single master password that opens all DB tables. A user with this password need see only one password form to gain access to all DB tables. Or you may set unique passwords for particularly sensitive DB tables.

Note In addition, auxiliary passwords are supported by DB tables but you cannot access this feature from IntraBuilder. Auxiliary passwords allow you to create multiple individual passwords for each DB table, so that you can restrict access to certain tables and certain fields. Different users can be given different passwords that will open only a specific set of DB tables or allow read/write access to only certain fields within those tables. However, to set auxiliary passwords for field rights to a DB table you must use Paradox.

The process of assigning passwords is initially very similar to that described previously for DBF tables. To assign a master password to a DB table, follow these steps:

1. Make sure the DB table you want to secure is closed.
2. From the File menu, select Database Administration. The Database Administration dialog box appears.
3. Make sure that the Current Database field is set to <None> and the Table Type field is set for Paradox (DB) tables.
4. Click the Security button to open the Security dialog box.
5. Select the name of the table in the Table list. If the table is not in the current directory, use the Folder button to select the directory.
6. Click the Edit Table button to open the Master Password dialog box.
7. Enter the new password for the table in the Master Password field. The password can be up to 31 characters long and can contain spaces. Paradox passwords are case-sensitive.
8. Enter the password again in the Confirm password field.
9. Click the Set button to save the password.

Removing passwords from DB tables

Related topics

To remove an existing password from a DB table, follow Steps 1 through 6 in the previous section. When prompted, enter the existing master password for the table. Then click the Delete button to remove the password from the table.

Custom security

Related topics

IntraBuilder's extended JavaScript offers a range of security options. Using JavaScript, you can always create your own custom login form. There are a number of reasons to do this, depending on the security strategies you choose. You might want to ask for more (or less) information than the automatic password form does.

For example, you might want to limit all read/write access to local network users and restrict Web-users to read-only. In that case, for Web access you could preset the form with the same user name and require the user to type only the password. If the password matches, the user gets the predetermined level of access, in this case, read-only. If someone stole the password, the intruder would still get only the read-only level of access because the user name is hard-coded into the form. This strategy would work for DBF and SQL tables (of DB tables with auxiliary passwords set by using Paradox).

If you wanted to offer higher-level security, such as read/write access, over the Web, you might want to create a second, separate non-publicized version of the form with a password login. You could use either the automatic password form or write a custom login form requiring both user name and password, for users authorized to edit the tables. Average users, knowing nothing of the higher-security level of the form, would simply get immediate read-only access to the preset version of the form without having to go through any login at all.

Another reason to write a custom login form: you might want to replace your Web server's login security provision or to augment it. With JavaScript, the possibilities are endless.

For an example of a practical, working custom login form, see the multi-page login form for the Threaded Message Database (TMD) project in Customizing the application.

Use the Password form component when prompting for a password in a custom login form. This data-entry field displays asterisks rather than readable characters while entering, thus obscuring the password from any observers when the form is run on a browser, but not when the form is run locally. See Working with components for more on the Password component.

The Threaded Message Database project

Related topics

Instead of theorizing in a vacuum, the best way to get a sense of what IntraBuilder can do and how it works is to actually try to make something. In the process of using the tools to build any real-world application, you will see all the important basic concepts and many of the intermediate techniques that are used to build a Web-based database application.

As a learning exercise, we will construct over the course of the next several chapters, an intranet application called Threaded Message Database (TMD). This application more resembles CompuServe and BBSs as opposed to the Usenet. A thread of messages consists of an original message, all the replies to that message, all the replies to the replies, and so on. By the time it's finished, you will be able to navigate through the thread, skipping all unrelated messages.

Typing in the project code

Related topics

The JavaScript language is case-sensitive. This means that not only must you spell the code correctly, but you must also capitalize it exactly as shown.

There are also a few places where the SQL language is used. SQL is not case-sensitive. In this guide, the code is capitalized specifically to differentiate language keywords (in lowercase) from table and field names (in uppercase).

Creating a project directory

Related topics

Create a directory named TMD under the IntraBuilder\Apps subdirectory with the Windows Explorer or through IntraBuilder's Script Pad, for example:

```
_sys.os.makeDir( "C:\\PROGRAM FILES\\BORLAND\\INTRABUILDER\\APPS\\TMD" )
```

Make the TMD directory your current directory.

Getting the latest information

Related topics

Be sure to get the latest source code for the TMD project from Borland Online at http://www.borland.com/techpubs/intrabuilder.

Accessing tables introduction

Related topics

This series of topics introduces basic JavaScript concepts for accessing tables and provides a procedure for connecting IntraBuilder to remote databases, including industry-standard SQL servers.

For information on restricting access to tables for security, see Security.

Data access object overview

Related topics

IntraBuilder's advanced, event-driven data model is implemented entirely in a handful of classes:

- Session
- Database
- Query
- Rowset
- Field

There is also a StoredProc class for calling stored procedures in SQL-server databases. The StoredProc class is parallel to the Query class in the class hierarchy.

These classes are described in detail in the *Language Reference*. This series of topics is intended to give you a sense of how these classes fit together, and to give you a more general perspective. As is sometimes the case, the best place to start explaining these classes is the middle.

Query object

Related topics

Query objects are the center of the data model. If you want to access a table, you must use a query. When you drag a table from the IntraBuilder Explorer and drop it onto your form or report in the designers, you create an active Query object for that table on your form.

A Query object's main job is to house two important properties, *sql* and *rowset*.

sql property

Related topics

The *sql* property contains a SQL SELECT statement that describes the data to be obtained from the table. In other words, the SQL SELECT statement specifies which tables to access, any tables to join, which fields to return, the sort order, and so on. This information is what many people think of when they hear the word query, but in IntraBuilder, SQL statements form one of many properties of the Query object.

SQL is a data sub-language. It is not a full-fledged programming language, but rather a standard, portable language designed to be used in other language products to access databases. In this regard, SQL has succeeded; it is the industry standard, and IntraBuilder uses SQL for its intended purpose, to access data.

When you use IntraBuilder Experts and the drag-and-drop capability of the IntraBuilder Designers, IntraBuilder builds the SQL statement for you. Once a table has been accessed by the SQL statement, you can do almost anything you want with IntraBuilder's data access objects, including navigating, searching, editing, adding, and deleting. You can create complete Web-based database applications without knowing a word of SQL.

On the other hand, knowledge of SQL is a good thing. It can be used to access the entire spectrum of relational databases, from DBF and DB tables on a local disk to advanced database servers. The power of IntraBuilder lets you learn it at your leisure.

The *sql* property is a string containing a SQL SELECT statement, such as:

```
select * from BIOLIFE
```

The * means all the fields and BIOLIFE is the name of the table, so that statement would get all the fields from the BIOLIFE table.

rowset property

Related topics

When a query is activated by setting its *active* property to true, the SQL statement in the *sql* property is executed. If there are no errors, the SQL statement generates a result: a set of rows, or rowset.

Rows are sometimes referred to as records, but to be more precise, a record is the physical row in a single table. If the SQL SELECT statement simply gets all the fields from a single table, then each row is a record. But with anything else—fewer fields, more fields, more tables—the rows that are generated can no longer be considered records.

Because a Query object contains only one *sql* property to describe the data, it contains only one *rowset* property, which refers to the Rowset object that represents the results.

Rowset object

Related topics

Although the Query object may be the center of the data model, the Rowset object is where all the action is. While you must use a query to get access to data, you must use the query's rowset to do anything with the data.

The row cursor and navigation

Related topics

The rowset maintains a row cursor—not to be confused with the many other cursors used in computers—which points to the current row in the rowset. When the query is first activated, the row cursor points at the first row in the rowset.

You can get and store the current position by calling the rowset's *bookmark*() method.

To move the row cursor, call the rowset's navigation methods:

- *next*() moves the cursor a specified number of rows relative to its current position.
- *first*() goes to the first row in the rowset.
- *last*() moves to the last row.
- *goto*() uses the value returned by *bookmark*() to move back to that specific row.

Because each rowset maintains its own row cursor, you can open multiple queries—each of which has its own rowset—to access the same table and point to different rows simultaneously.

Rowset modes

Related topics

Once a query has been activated, its rowset is always in one of the following five modes:

- Browse mode, the default, which allows navigation only.
- Edit mode, which allows changes to the row.
- Append mode, in which the user can type new values for a row, and if the row is saved, a new row is created on disk.
- Filter mode, used to implement Filter-By-Form, in which the user types values into the form and IntraBuilder filters out all the rows that do not match.
- Locate mode, similar to Filter mode, except that it just searches for the first match, instead of setting a filter.

Rowset events

Related topics

A rowset has many events used to control and augment its methods. These events fall into two categories:

- can- events—so named because they all start with the word can—which are fired before the desired action to see whether an action is allowed to occur; and
- on- events, which fire after the action has successfully occurred.

Row buffer

Related topics

The rowset maintains a buffer for the current row. It contains all the values for all the fields in that row. The buffer is accessed through the rowset's *fields* property, which refers to an array of Field objects.

Field objects

Related topics

The rowset's *fields* array contains a Field object for each field in the row. In addition to static information, such as the field's name and size, the most important property of a Field object is its *value*.

value property

Related topics

A Field object's *value* property reflects the value of that field for the current row. It is automatically updated as the rowset's row cursor is moved from row to row.

Assigning a value to the *value* property changes the value in the row buffer, and if the row is saved, those changes are written to disk.

Important When referring to the contents of a field, don't forget to use the *value* property. For example,

```
this.form.rowset.fields[ "Species" ].value
```

If you leave out *value*,

```
this.form.rowset.fields[ "Species" ]
```

you are referring to the Field object itself, which is rarely intentional—except for *dataLink*s, explained next. Get in the habit of including *value* when referring to a field; if you don't, the code doesn't work.

Using dataLinks

Related topics

Just as a Field object's *value* property is linked to the actual value in a table, a visual component on the form can be linked to a field object, through the form component's *dataLink* property. This property is assigned a reference to the linked Field object. When connected in this way, the two objects are referred to as *dataLink*ed.

As the rowset navigates from row to row, the Field object's *value* is updated, which in turn updates the component on the form. Changes to the form component are echoed in the *dataLink*ed Field object, which in turn are saved to the table.

Database objects

Related topics

Now going one level up in the object hierarchy from queries are Database objects. These objects have three main functions:

- To give access to a database
- Database-level security
- Database-level methods

Accessing a database

Related topics

A Database object is needed to access SQL servers and ODBC databases. The Database object's *databaseName* property is set to the BDE alias for the desired database. The BDE must be setup to access the database before using IntraBuilder.

Database-level security

Related topics

Many SQL servers and ODBC databases require the user to login to the database. The Database object's *loginString* property can be preset with a valid user name and password to login to the database automatically.

Because each Database object represents access to a database, you can have multiple Database objects that are logged in as different users to the same database.

Database-level methods

Related topics

The Database object contains methods to perform database-level operations such as transaction logging and rollback, table copying, and re-indexing. Different database formats support each method to varying degrees.

Default database

Related topics

Each session includes a default database for accessing DBF and DB tables. This default database does not have a BDE alias. When you create a query object, it is initially assigned to the default database. If you're accessing DBF or DB tables, you don't need to create a Database object.

If you're accessing other table types, you'll need to create the Database object, assign the BDE alias, then assign the Database object to the Query object's *database* property. This must be done before attempting to activate the query.

Session objects

Related topics

At the top of the object hierarchy is the Session object. Each session represents a separate user, and is used primarily for the session-level security used by DBF and DB tables. Security and sessions are discussed in Security.

StoredProc objects

Related topics

StoredProc objects are used to call stored procedures in SQL-server databases. When calling a stored procedure, the StoredProc object takes the place of the Query object in the class hierarchy; it is attached to a Database object that gives access to the database, and it can result in a Rowset object that contains Field objects.

The StoredProc object's *procedureName* property is set to the name of the stored procedure. Any parameters that are passed to the stored procedure are set in the *params* array. The stored procedure can either return values, which are read from the *params* array, or a rowset, which is accessed through the *rowset* property.

Connecting your IntraBuilder application to SQL servers

Related topics

IntraBuilder Professional and IntraBuilder Client/Server provide connectivity to industry-standard SQL database management systems, through the Borland Database Engine (BDE) configured with the corresponding Borland SQL Links drivers.

IntraBuilder Professional includes SQL link drivers for Borland InterBase and Microsoft SQL Server. IntraBuilder Client/Server includes a complete set of SQL Links drivers for Oracle, Sybase, Informix, IBM DB2, Microsoft SQL Server, and Borland InterBase. If you want to write custom SQL Links drivers to support other SQL database systems, contact Borland.

To connect your IntraBuilder application to a SQL database you need to configure BDE and your SQL Links Driver to access your SQL database. In this procedure you create an alias that BDE uses to locate the SQL database. Then you add this alias to the Database object on your IntraBuilder form.

Consult the documentation for your SQL database management system product for specific guidance on the initial steps of the following general procedure. Each product is a little different.

Procedure for connecting to SQL servers:

1. Make sure you have properly installed the client software for the database management system product to which you wish to connect (Oracle, Sybase, Informix, Borland InterBase, IBM DB2, or MS SQL Server).
2. Define server names or other connection strings in the product's required configuration files (for example, in Oracle, TNSNAMES.ORA and in Sybase, SQL.INI, and so on).
3. Test the connection by using the database vendor's connection utility (such as Sybase's SYBPING.EXE). If you cannot "ping" the server with this utility, probably BDE and IntraBuilder will not be able to access it either.
4. Make sure BDE and the Borland SQL Links product is properly installed. These core products are included in IntraBuilder Professional. If properly installed, the SQL Links drivers for Oracle, Sybase, MSSQL, Informix, and InterBase appear on the Drivers page of the BDE Configuration Utility (BDECFG32.EXE).
5. In BDECFG32.EXE, add an alias for the SQL server. Settings for the alias will vary according to vendor.
6. In IntraBuilder, open the IntraBuilder Explorer, click the Tables tab, then choose the SQL server alias from the drop-down menu in the Look In box (at the top of the IntraBuilder Explorer). You are then prompted for a login name and password to connect to that SQL server database. Once you connected successfully, you will see the tables in that database in the IntraBuilder Explorer.

The easiest way to use a table in a SQL server database in a form or report is to drag the table from the IntraBuilder Explorer onto the surface of the form or report in the Form or Report designer. This automatically creates the Database object required to connect to the database, and the Query object for that table.

IntraBuilder offers alternative ways of doing things. You can also create the Database object in a script or drag a Database component from the Component Palette to the design surface and set its databaseName property to the alias you created in BDECFG32.EXE.

The TMD project: Designing the Messages table

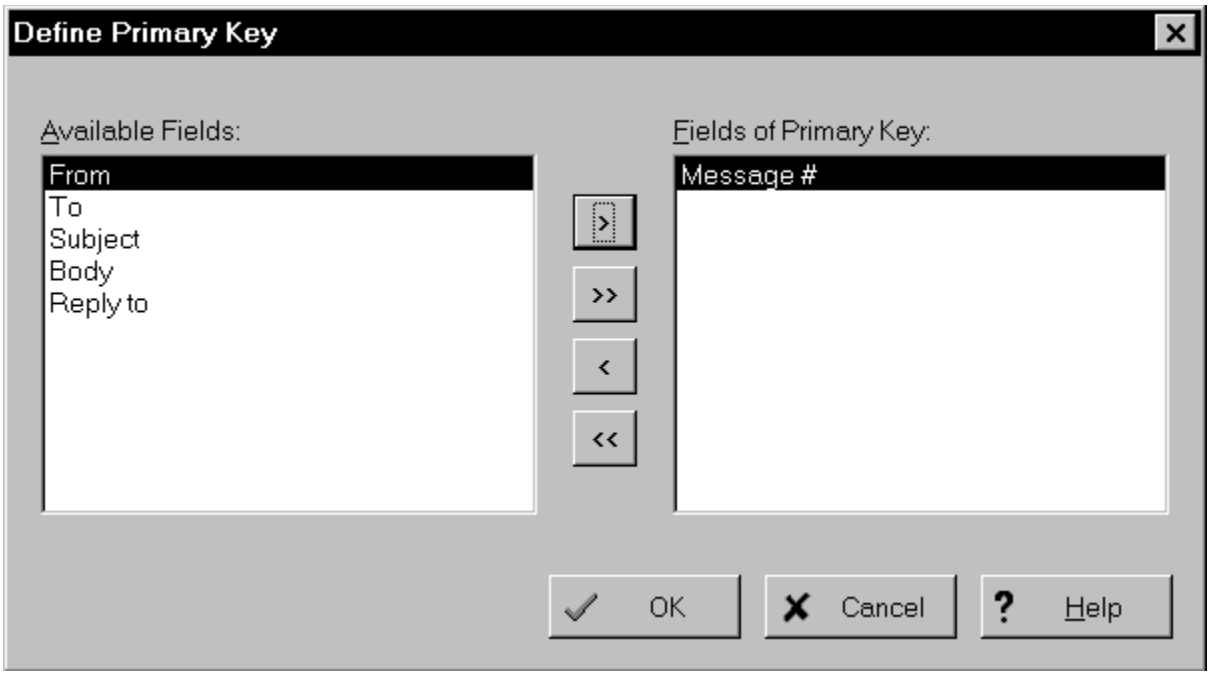
Related topics

The main table for the Threaded Message Database project is the table of messages. To create this table:

1. Make sure you're in the TMD directory you created in Creating a project directory.
2. In the Tables tab of the IntraBuilder Explorer, double-click the (Untitled) table.
3. At the Expert or Designer prompt, choose Designer.
4. Make sure the table type is Paradox.
5. Create the following table structure:

Name	Type	Width
Message #	AutoIncrement	4
From	Alpha	30
To	Alpha	30
Subject	Alpha	30
Body	Memo	10
Reply to	Long	4

6. In the Structure menu, choose Define Primary Key.
7. Select the Message # field as the only primary key field, click the right arrow to add it to the list of primary key fields, and click OK.
8. Save the table as MESSAGES.DB and close the Table Designer.



JavaScript forms

Related topics

This series of topics details the structure and behavior of the JavaScript forms generated by the IntraBuilder Form Expert and Form Designer.

Understanding the structure of JFM files, which the Form Designer generates, is the key to using two-way tools effectively. To understand the structure you need only create a simple form, as described in the next topic.

Creating a simple form

Related topics

Use the Form Designer to create a simple form:

1 Make sure you're in the TMD directory, which contains the MESSAGES.DB file you created in The TMD project: Designing the Messages table.

2 In the Forms tab of the IntraBuilder Explorer, double-click the first (Untitled) form.

3 At the Expert or Designer prompt, choose Designer.

4 Drag the MESSAGES.DB table from the IntraBuilder Explorer onto the form surface. This creates a Query object named messages1.

5 Drag a Button control from the Component Palette onto the form. This creates a Button object named button1.

6 Right-click button1 and choose Inspector from the shortcut menu.

7 In the Inspector's Events tab, click the *onServerClick* event.

8 Click the tool button on the right to create an event handler.

9 In the Method Editor, type in the following code so that the method appears like this:

```
function button1_onServerClick()
{
  this.text = this.form.rowset.count();
}
```

10 In the method selector in the top left corner, choose "(Header)". This displays the script's header, which starts out blank. Type in the following:

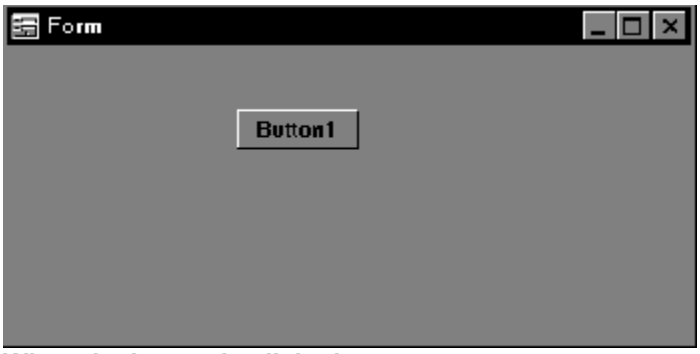
```
// Anatomy.jfm
//
```

11 Close the form. This brings up the standard Changes Made dialog box. Click Yes. Save the form as ANATOMY.JFM.

Run the form.

The ANATOMY form counts the number of rows in the Messages table and displays that number on the button. When you click the button, the number 0 appears because there are no records in the table yet. Now close the form.

The ANATOMY.JFM form:



When the button is clicked...

Examining the generated code

Related topics

Right-click ANATOMY.JFM in the Explorer, and choose Edit as Script from the shortcut menu. This opens the script in the Script Editor. The numeric properties will probably be different, but the script should look like this:

```
// Anatomy.jfm
//
// {End Header} Do not remove this comment//
// Generated on 08/08/1996
//
var f = new AnatomyForm();
f.open();
class AnatomyForm extends Form {
   with (this) {
      height = 9.2;
      left = 20.5;
      top = 10.4;
      width = 53.5;
      title = "Form";
   }
   with (this.messages1 = new Query()){
      left = 15.625;
      top = 3.95;
      sql = 'SELECT * FROM "messages.db"';
      active = true;
   }
   with (this.messages1.rowset) {
   }
   with (this.button1 = new Button(this)){
      left = 18;
      top = 2;
      width = 9.5;
      text = "Button1";
      onServerClick = class::button1_onServerClick;
   }
   this.rowset = this.messages1.rowset
   function button1_onServerClick()
   {
     this.text = this.form.rowset.count();
   }
}
```

JFM file structure

Related topics

There are four major sections in a JFM file:

- The first part is the optional Header section. This is any code above the // {End Header} line. Comments that describe the file are usually put here.
- Between the header and the beginning of the *class* definition is the standard bootstrap code. This code instantiates and opens a form when you run the form through *_sys.forms.run*(), similar to the way the boot sector of a disk starts the system when you turn on your computer.
- The main *class* definition constitutes the bulk of most JFM files. This is the code representation of forms designed visually in the Form Designer. Note that this is a subclass of the Form class.
- Everything after the main class definition, if anything, makes up the General section. This is a place for other functions and classes, but is rarely used.

Form class definition

Related topics

Like any other *class* definition, the main one in the JFM can be further broken down into two parts:

- The constructor is the code that is run every time a *new* object of that class is instantiated. It creates, or constructs, an object of that class. Class constructors created by the Form Designer are divided into four parts:
- Assignments to the stock properties of the Form object. This is the single *with*(*this*) block at the beginning.
- Data access objects in the form, each with its own *with* block.
- All the controls in the form, each with its own *with* block.
- Housekeeping code; specifically to assign the rowset of one of the queries in the form to the form's *rowset* property as the form's primary rowset.
- Class methods, if any, follow. This is usually event handler code, but can also contains other methods that pertain to the form and are often called by the event handlers.

How the contents are generated

Related topics

The contents of the class constructor are the direct result of the visual development environment: the position of the controls and the other stock properties of the form and its contents. You can create and edit class methods from the Method Editor in the Form Designer. Both the header and general sections are also editable from the Method Editor. You have no control over the bootstrap code generated by the Form Designer.

Editing a JFM

Related topics

If you're going to be doing some serious text editing of a JFM file, the Method Editor seems a bit cramped. You can use any text editor, like the built-in Script Editor (which features color syntax highlighting), or your favorite programmer's editor.

Be sure to save and close the JFM file before editing the form in the Form Designer. No matter which tool you use, you want to preserve the Two-Way nature of the Form Designer so that any changes you make manually will not be lost the next time you save the form from the Form Designer.

Editing the header and bootstrap

Related topics

The first “safe JFM” rule involves the line that says:

```
// {End Header} Do not remove this comment//
```

Don’t remove or modify it! If you do, you might lose the contents of the header or further confuse the Form Designer.

The next rule is about the standard bootstrap code: don’t bother changing it. Every time the JFM is written the same standard bootstrap is rewritten anew, so any changes you make will be lost.

If you want to change the way the form is instantiated and opened when you run the form, instead of changing the bootstrap code, you need to add to it or replace it by placing your own bootstrap code in the header.

The key is to realize that a JFM is just a JS script file with a fancy extension. When you run the form, the code at the top is run just like when you run a script. To put it another way, there is nothing magical about the standard bootstrap code—it just happens to be the first code that is found at the top of a plain JFM file. If there are some comments in the header, as in the sample file above, they have no effect.

You can place any code you want in the header. The Form Designer will ignore it. See Custom header code for an example of using a custom header.

Editing properties in the JFM

Related topics

Inside a *with* block, you may assign values to existing properties only. Therefore, you are free to edit the values assigned to any of the properties in the class constructor, or add assignments to the objects' stock properties.

Most properties must be of a particular data type. For example, *pageno* is a number and *sql* is a string. If you change the property, you must maintain the correct type.

One notable exception is the *value* property. If a component is *dataLink*ed to a field, the type of that field determines the type of the *value*. But if the component is not *dataLink*ed, its type can by any of the simple data types. In the Inspector, you can use the type button to select the type of the value you're assigning to the property if the property can accept multiple types.

The Form Designer leans toward literals as opposed to expressions. For example, suppose you want a Text component to default to the current date. You could edit the JFM so that the assignment reads:

```
value = new Date();
```

That would work fine up until the next time you edit the form in the Form Designer. The expression gets evaluated when the form is loaded so that the *value* property has an actual date. Then that date gets saved to the JFM file which causes the date that you last edited the form is hard-coded into the form.

The simplest way to solve the problem is to set the *value* property programmatically, which puts it outside the reach of the Form Designer. The most convenient place is the component's *onServerLoad* event. A simple codeblock like:

```
{; this.value = new Date()}
```

does the trick. When the form is run, the form's *onServerLoad* and each component's *onServerLoad* event, if any, is called in turn. This codeblock updates the *value* to the current date. The Form Designer knows that a codeblock is attached to the *onServerLoad* event, and obediently reads and writes it, but it doesn't know what's inside it, and doesn't change it.

The TMD project: Generating the Viewer form

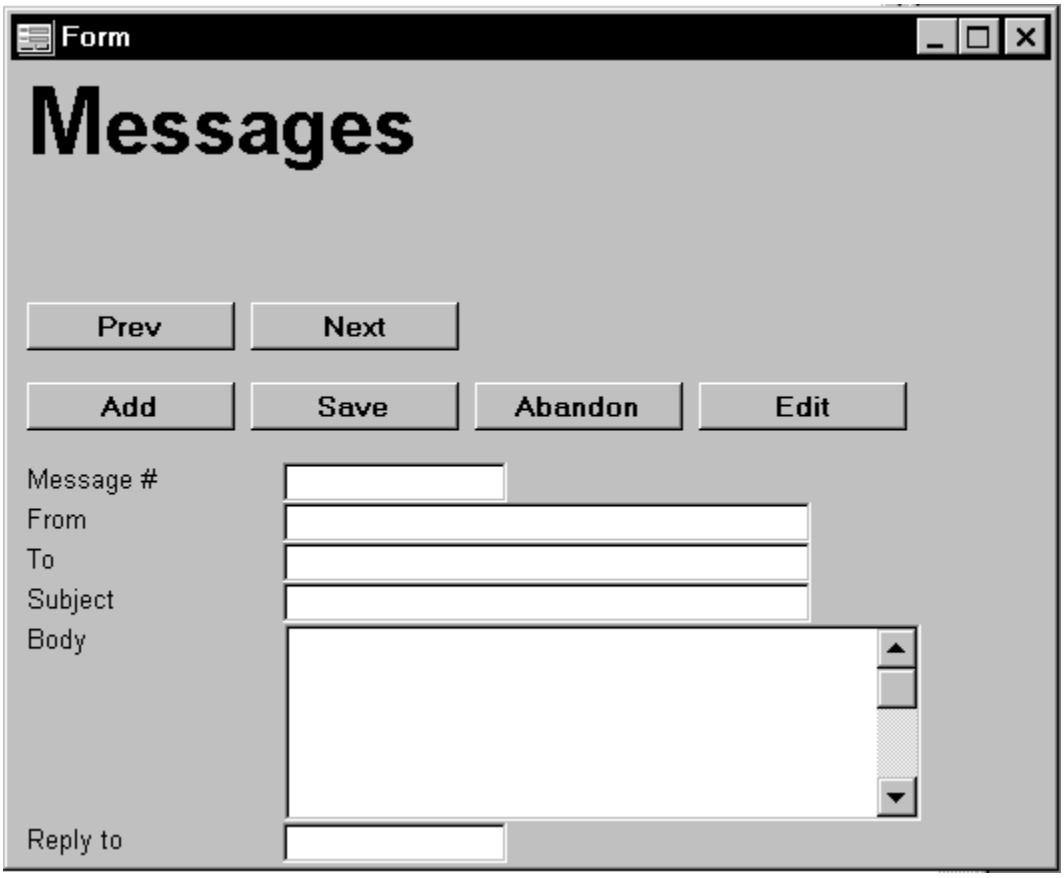
Related topics

The main form for the Threaded Message Database project is the Viewer form. It allows the user to browser through the messages, and compose new ones.

Create the form with the Form Expert by following these steps:

1. In the Forms tab of the IntraBuilder Explorer, double-click the first (Untitled) form.
2. At the Expert or Designer prompt, choose Expert.
3. Choose MESSAGES.DB from the list of tables and click Next.
4. Click the >> button to select all the fields and click Next.
5. Choose columnar layout and click Next.
6. Leave the default scheme and click Next.
7. Choose the Buttons control type, and choose Top in the “Location on Form” box.
8. Select the following buttons:
   - Next
   - Previous
   - Add
   - Edit
   - Save
   - Abandon

   and click Next.
9. Click Run Form.
10. In the Save Form dialog box, save the form as VIEWER.JFM.Take a good look. Now close the form.



Database access from forms

Related topics

This series of Help topics takes you through the process of adding basic database functions to an IntraBuilder form, starting with the Expert-generated form for the Threaded Message Database (TMD) application that you created earlier.

The basic database functions are:

- Displaying data
- Navigating the rowset
- Editing existing rows
- Validating data
- Adding new rows
- Filtering rows
- Locating rows
- Deleting rows

Each section discusses the database access objects used to implement the function and how they tie into forms. In the process of learning these general tasks, a number of side issues are examined.

You will modify the TMD project form as each topic is addressed. By the end of this series of topics, you will have a basic, functional TMD application. This project will be further enhanced later.

Displaying data

Related topics

The first and most basic job of a form is to display data from a table. To see how this is done, examine a component that displays data.

For example, in the Viewer form, a Text control displays the subject of the message. In the Form Designer, inspect the Text object. On the Properties tab the *dataLink* property contains something like:

```
parent.messages1.rowset.fields[ "Subject" ]
```

This is a reference to the Field object which represents the Subject field in the table.

Object references

Related topics

Object references like those shown in the Form Designer or used in event handlers, are always relative. Periods define the hierarchy of objects, somewhat as slashes define the hierarchy of files in the Windows file system.

For example, here is the hierarchy of the Text object in the TMD project:

- The Text object's *parent* is the form. In other words, the Text object is contained in the form.
- The form contains a Query object that gives access to the table. A reference to this Query object has been assigned to a property of the form named messages1 by the Form Expert, and it hasn't been changed.
- messages1, like all Query objects, has a *rowset* property which refers to the query's Rowset object, which in turn represents the results of the query.
- The Rowset object has a *fields* array, which contains a Field object for every field in the rowset.
- Because the *fields* property refers to an array, its elements are referred to by using the square brackets instead of the dot operator. The fields in the *fields* array are accessed by name with a string; in this case, "Subject."

Other fields in the same query are accessed in the same way, with the corresponding field name in the brackets. Of course, you often don't have to type all of this; both the Expert and the Choose Field tool in the Form Designer generate it for you. The components in the Field Palette are also all linked to fields in the table by means of preset *dataLink* properties, as explained in the next section.

dataLink and value properties

Related topics

You link form components to a table's field by assigning the Field object to the *dataLink* property of the form component; this is conveniently called datalinking.

Both field and component objects have a *value* property. Changes in one object's *value* property are echoed in the other. The form component's *value* property reflects the value displayed in the component at any given moment. If the component's value is changed, it is copied into the field, either after the component loses focus (if you're running the form locally in IntraBuilder) or when the entire form is submitted (if you're running the form remotely on a browser).

The *value* property for all fields in a rowset are set when you first open a query and updated as you navigate from row to row. The *value* properties for components *dataLink*ed to those fields are also updated at the same time, unless the rowset's *notifyControls* property is set to false. You can also force the components to be updated by calling the rowset's *refreshControls*() method, which is useful if you have set the a field's *value* property through code.

Other field properties

Related topics

Field objects have events like *canChange* and *beforeGetValue* that can affect how *value* properties are updated. These are discussed in Two-way field morphing.

Displaying data in a form

Related topics

The Form Expert creates *dataLink*ed components for the requested fields. If you're creating a form from scratch or simply want to add another field to an existing design, the easiest way is to drag the field from the Field Palette. The Field Palette displays all fields from all active queries. If there is no active Query object on the form, the Field Palette is empty.

The easiest way to create an active query to access a table is to drag the table from an Explorer—Windows' or IntraBuilder's—onto the form. You can also create a query manually by dragging a Query object from the Component Palette (Data Access tab), setting its *sql* property, and then setting its *active* property to true.

You can also manually create a *dataLink*ed component by dragging it from the Component Palette and setting its *dataLink* property. But remember to add and set a Query object first; you can set the *dataLink* property only to a field in an active query.

You should also change the *name* property to something more descriptive than the default Text1, Text2, and so on. This is especially important if you might refer to the component in code. A descriptive name will be much easier to remember. Change the name as soon as possible, before you create any event handlers, because the Form Designer uses the component's *name* when creating the function, and you'll want descriptive function names, too.

Although *dataLink*s are the easiest and most straightforward way to display data, there are other ways. In particular, you can use an HTML object and update it during the *onNavigate* event.

Navigating the rowset

Related topics

Navigating the rowset is done through methods and events of the rowset object. In fact, everything in this series of topics besides simple data display is done through methods and events of the rowset object.

It's important to remember that the *rowset* is a property of the query. A reference to the query is usually readily available; it's stored as a property of the form. Don't forget to include the *rowset* property.

For example, this would not work:

```
this.form.query1.next();
```

This would work:

```
this.form.query1.rowset.next();
```

A form also has a primary rowset, referred to by the form's *rowset* property. This is especially useful if the form contains only one query—and therefore one rowset. You could then use a shortcut like:

```
this.form.rowset.next();
```

Basic navigation

Related topics

Basic navigation means the user’s ability to move to the next row, the previous row, the first row, and the last row.

Relative row navigation

Related topics

All relative row navigation—the next row, the previous row, two rows ahead, 100 rows behind—is accomplished through the rowset's *next*() method. This method takes a single optional numeric parameter that indicates how many rows to move and in which direction, positive or negative. If no number is specified, the default is 1, which means one record forward; the next row. To move to the previous row requires a parameter of –1.

The *next*() method respects any active filter conditions. As a simple example, suppose you have three rows and you're currently on the first one. If the second row is filtered out because it does not meet the filter condition, a simple *next*(), which moves forward one row, would move the row cursor to the third row.

Filters are explained in Filtering rows.

If you run out of rows—for example, you're on the last row and try move forward one, or you're on the 20th row and try to move backward 100—the row cursor moves to the end-of-set and stops.

First and last rows

Related topics

To go to the rowset's first or last row, call the rowset's *first*() or *last*() method. These methods also respect filters like the *next*() method. If there is only one row in the rowset or only one row in the rowset that matches the filter conditions, then *first*() and *last*() will both move you to that one and only row. If there are no rows in the rowset or no rows that match the filter conditions, *first*() and *last*() will move you to the end-of-set.

endOfSet property

Related topics

A rowset has two ends: one before the first row, and one after the last row. (If there are no rows in the rowset, then those two ends are technically the same—not that it matters, because you can't go anywhere else anyway.)

A rowset's *endOfSet* property contains true or false to indicate whether the row cursor is at either end-of-set or not; in other words, whether the row cursor is pointing to a row that contains data.

> If *endOfSet* is true, then the row cursor is at one end of the set—the property doesn't indicate which end—and any attempt to access the *value* property of a field generates an error, because there's no data there. (Note that *dataLink*ed components show blanks.)
>
> If *endOfSet* is false, then the row cursor is at an actual row with data.

The *next*(), *first*(), and *last*() methods all return true when they move the row cursor to a valid row. They return false if the navigation results in the row cursor being at the end-of-set. Because you usually don't want to see the blank rows at the end-of-set (sometimes referred to as phantom rows) you can use the return values to make relative navigation a little smarter.

To go to the next record,

```
if ( !this.form.rowset.next() ) {      // If gone past the last row to end-
of-set
  this.form.rowset.next( -1 );         //   go back to the last row
}
```

To go to the previous record,

```
if ( !this.form.rowset.next( -1 ) ) {  // If gone before first row to end-of-
set
  this.form.rowset.next();             //   go to first row
}
```

Notice that it doesn't really matter that the *endOfSet* property does not indicate which end of the set you're on, because you can infer that from the direction you moved. If you moved too far forward, that would take you after the last row, so you would want to go to the last row. If you moved too far backward, that would take you before the first row, so you would want to go to the first row.

If *first*() or *last*() return false, that means there are no rows in the rowset, or if a filter is active, at most no rows that match the filter. (If there are no rows, then setting a filter certainly won't improve the situation.) *endOfSet* would always be true.

Again, because you usually don't want to display the phantom row, check the *endOfSet* property when you first activate the query, when you set a filter, or after you delete a row. Then you can take specific action, like displaying a message that there are no rows or automatically give the user the opportunity to add new rows. This is explained in Adding rows to an empty table.

Looping through a rowset

Related topics

To visit each row in a rowset, start at one end and move through the rowset with the *next*() method, checking the *endOfSet* property to see if you've finished. This is usually done in the forward direction, as shown here:

```
this.form.rowset.first();
while ( !this.form.rowset.endOfSet ) {
  //
  // do whatever
  //
  this.form.rowset.next();
}
```

Because the *endOfSet* check is at the beginning of the *while* loop, it is checked immediately after the *first*() method is called. If there are no visible rows, the contents of the loop are never executed.

You might also want to set the rowset's *notifyControls* property to false, so that you don't waste time updating *dataLink*ed controls as you loop through the rowset.

Make sure to set it back to true after the loop is done. In fact, you might want to use a *try* block, so that even if there is an error during the loop, *notifyControls* is restored:

```
this.form.rowset.notifyControls = false;
try {
  this.form.rowset.first();
  while ( !this.form.rowset.endOfSet ) {
    // do whatever
    this.form.rowset.next();
  }
}
finally {
  this.form.rowset.notifyControls = true;
}
```

Navigation events

Related topics

Two events are the direct result of navigation: *canNavigate* and *onNavigate*.

- canNavigate event
- onNavigate event

canNavigate event

Related topics

Like all can- events, *canNavigate* fires when the navigation is attempted to see if it can occur. In other words, if there is a method assigned to the rowset's *canNavigate* property, it is fired when a navigation method like *next*() is called.

If the event handler method returns false, then the navigation simply does not occur; no error is generated, and no message is displayed unless you do so yourself in the *canNavigate* event handler.

If the event handler method returns true, then the navigation occurs. If there is no event handler method assigned to the property, that also allows the navigation to occur.

Use *canNavigate* to prevent navigation or to perform some action on a row just before you leave it. Because *canNavigate* doesn't know what kind of navigation is being attempted or the destination, it's usually not used to prevent navigation to a particular row, but rather to prevent the user from moving from the current row. This is often done because the current row contains invalid data and you want the user to fix it or abandon it, leaving the row in its previous and hopefully valid condition.

It's not necessary to use *canNavigate* for this purpose, however, because the *canSave* event will fire after *canNavigate* if the row was modified and needs to be saved. If the rowset's *modified* property is true, IntraBuilder will attempt to save the current row before navigating to another one (and that would be only after *canNavigate* returns true, which is why *canSave* fires after *canNavigate*, not before).

Use the *canSave* event to verify that the row is valid and can be saved. If *canSave* is called and returns false, then the navigation does not occur, even if *canNavigate* returns true. *modified* and *canSave* are explained in more detail in canSave event.

You will often use *canNavigate* to do something with a row just before leaving it. In this case, *canNavigate* performs its action and always returns true.

onNavigate event

Related topics

onNavigate fires after the navigation occurs, when the row cursor is at the final destination, either a valid row or the end-of-set. Use it for some action that must be performed as you navigate from row to row.

For example, instead of using a *dataLink*ed component to automatically display the contents of a field, you could use an HTML object to display—but not edit—a field. You would do this if you want to make the field read-only, or use the richer text formatting available through an HTML object. For an example, see Displaying unlinked data.

Because *onNavigate* fires even when moving to the end-of-set, and because attempting to access a field while at the end-of-set causes an error, most *onNavigate* event handlers include an *endOfSet* test.

These kinds of actions also should occur when the form is first opened. At that point, no navigation has occurred, so *onNavigate* is not called by IntraBuilder. You'll need to call the rowset's *onNavigate* event (or call the same code the *onNavigate* event handler does) from the form's *onServerLoad* event.

Navigating the rowset in a form

Related topics

The Form Expert provides the option of including buttons that do the basic navigation—next, previous, first, and last—in the form's primary rowset. These buttons call the corresponding methods and avoid displaying the phantom row at the end of the rowset, by using the code detailed in endOfSet property.

The Form Expert lets you choose between plain HTML buttons or images for the individual components. The images are consistent in size and more attractive, while the buttons are faster but vary in representation, depending on the browser.

Instead of separate components, the Form Expert uses a single image with navigation graphics and use the Image object's *onImageServerClick* event to see where the user clicked and call the appropriate navigation method. You can do the same.

You may create a form page that appears if there is no data.

Every time navigation occurs, after *dataLink*'d components are automatically updated and all pending events have fired, the updated contents of the form are displayed as HTML on the browser.

The TMD project: Navigating the rowset

Related topics

First and last are not significant in a message database. The Next and Previous buttons are the only practical choices. You added these buttons when you created the Viewer form by using the Form Expert.

Editing existing rows

Related topics

With *dataLink*ed components, IntraBuilder makes it easy to create data-entry applications that work over an intranet. Data from tables is displayed in components on a form in the browser. A user's changes in the component are echoed in the linked Field object, which in turn posts the data to the table.

You don't need to do much to enable editing. On the other hand, there are a number of features that offer you extensive control over how data is edited.

state and autoEdit properties

Related topics

Every rowset has a *state* property that indicates the rowset's current mode. Until a query is made active, its rowset's state is Closed; the rowset is inactive and has no data. When the query's *active* property is set to true, the SQL statement in the query's *sql* property is executed, which creates the access to the table and generates the data in the rowset.

At this point, the rowset's *state* depends on another property, *autoEdit*.

If *autoEdit* is false, then the rowset opens in Browse mode, but can be switched to Edit mode.

If the rowset's *autoEdit* property is true (the default), then the rowset opens in Edit mode, and cannot be switched to Browse mode.

The difference between Edit and Browse mode is that in Browse mode, the rowset is read-only. Form components *dataLink*ed to fields in a rowset in Browse mode attempt to display the data in a non-editable fashion (depending on the browser). For example, instead of displaying data in a Text component in which you can type, you might see the data as plain text. Even if the browser displays the field in a form component that does allow editing, no changes are echoed in the *dataLink*ed Field object.

In Edit mode, the user can change the field values in the rowset; the browser uses the normal editing components on the form. Therefore, the simplest way to enable editing is to set the rowset's *autoEdit* property to true, which automatically allows editing of fields in *dataLink*ed components, without having to explicitly switch to Edit mode.

Browse and Edit modes are intended to control how data appears on the browser. When assigning values to fields in server-side code, the rowset is automatically switched from Browse mode to Edit mode if necessary.

Switching to Edit mode

Related topics

The rowset's *autoEdit* property is true by default, but you can set it to false to require a conscious switch to edit mode, rather than allowing automatic editing. The reason for this stems from the difference between traditional direct-access data-entry, like over a LAN, and the remote-access nature of the Web.

The nature of Web data access

Related topics

With a traditional LAN-based data-entry application, you have direct and dedicated access to your data. Such database applications know the instant you modify a displayed item of data. Because you've started to change a record, the database application can lock the row immediately, thus preventing others from changing the same record at the same time. Although you might switch to another application, the database application will maintain your partially edited data. As with any other application, if you try to close and exit it, it will make first sure that you've finished editing, by automatically saving, or asking you to save, or some other option.

In contrast, when a user accesses data through a Web browser, it's likely the user will at some point want to switch to a completely different location. Because this is done through the browser, fundamentally there's nothing you can do to stop it. As far as IntraBuilder is concerned, the user simply never responded, and the connection will eventually time out. If you're using *autoEdit* and the user changes some data and then goes to another Web page, the changes will not get posted.

Therefore it's often better to make users consciously choose to edit data, allow them to make the edits, and then consciously save the data. They can get confirmations on their save and know it's safe to go browse somewhere else. This strategy also makes accidental changes less likely, making it especially important for the broader class of less experienced users that will now have access to your data through the Web.

On the other hand, if the application is intended for well-trained people over an intranet, then you may allow automatic editing, because that is more natural and efficient.

Because editing occurs in the browser and the browser does not maintain a constant connection, if you use automatic editing, IntraBuilder cannot know if and when the user starts to edit data. This bit of information is necessary for active row locking, which is often used in traditional data-entry applications. (Locking is discussed in Row locking.) By using an actual switch to Edit mode, the user is declaring that he or she is changing data and the row should be locked.

Note *autoEdit* has no effect when running a form within the IntraBuilder Designer.

beginEdit() method and events

Related topics

To switch from Browse to Edit mode, call the rowset's *beginEdit*() method. When *beginEdit*() is called, the rowset's *canEdit* event is fired.

> If there is a method assigned to this event and it returns false, then the switch to Edit mode simply does not occur. No error or message is generated unless you do so yourself.
>
> If the event handler returns true or there is no event handler, then the rowset is switched to Edit mode. After the switch to Edit mode, the rowset's *onEdit* event is fired.

beginEdit() has no effect at the end-of-set.

Use *canEdit* to make sure that the user is allowed to edit the row. For example, in a message database, you may want to allow only the original author to change the contents of a message. This would require that the users identify themselves when they access the database; in fact you'll do this later in the TMD project.

If you want active row locking, try to lock the row in the *canEdit*. If the lock attempt fails, then program *canEdit* to return false.

You can use *onEdit* for things like recording when someone edits a row, or making copies of the initial field values so that they can be saved in a log file.

Row locking

Related topics

Use locks to manage simultaneous access to data. Only one user may have a lock on a particular row at any time.

Locks are always used at some point when editing data. Even if there's only one user, locks are still used; like making reservations at an unpopular restaurant, it doesn't hurt and they're easy to get.

There are two types of locks: active and passive. They differ in when they occur and how long they last. Passive locks expect rows to be available if not immediately, then within a few seconds. Active locks defeat this by holding on to locks for extended periods of time. Because of their opposite natures, you should not mix active and passive locks on the same table.

Passive locks (optimistic locking)

Related topics

With a passive lock, the row is locked momentarily when posting changes. This happens by default and prevents two or more users from changing the same row at exactly the same time. After the change is posted, the lock is released.

Because the row is locked only at the end of the edit, it's possible for one person to be making changes to a row, and for a second person to go to the same row, edit it, and save their changes before the first person has finished. When the first person saves their changes, the second person's changes are lost.

Note that with passive locks, *when* the two users start to edit is not relevant, even if they had to call *beginEdit*(). Note that *beginEdit*() only puts the rowset in Edit mode; it doesn't guarantee that an edit will take place. Everyone has their own version of the row on their browser. When and if they post is the deciding factor.

This type of mix-up might not matter. For a simple data-entry application, it would have been as if the first person had changed the data later anyway. But suppose that the application is reserving tables in a restaurant. The first person starts to reserve a particular table, but because the row is not locked, the second person starts and finishes the reservation, which is then lost. That wouldn't work. You would need to use active locks.

Active locks (pessimistic locking)

Related topics

By attempting to lock the record to begin editing, you ensure that only one person can edit a row at any given time. If a user cannot get the lock, he or she cannot edit the row.

To get an active lock, call the rowset's *lockRow*( ) method, as explained in the next section.

Lock attempts fail for two main reasons: Someone else already has a lock, or the application already has the maximum number of allowed locks. If a row has an active lock, passive locking is unnecessary and does not occur.

Consider a situation where the first user is viewing but not editing a row while the second user makes changes. When the first user tries to make changes, she sees that the row has changed; she can't do what she wanted. Again, the restaurant reservation is a good example. Because the first user was not editing the row, there was no active lock.

This is another reason why using *beginEdit*() to switch to edit mode is better than automatic editing, because the row will be updated when it is displayed for editing. With *autoEdit*, users would go through the motions of making the edit first but when they try to post their changes, the results might be unsatisfactory. The first user might override the second, even though the second had already made reservations; or the first user might have wasted time doing the edit to no effect.

Here is a fairly drastic but sure-fire solution for this type of application: Try to lock the row when you navigate to it, in the *onNavigate* event. If you can't get the lock, you can still display the row, but you would also display a message that someone else is already looking at it. One hitch with this approach is that if someone abandons the form—goes to another location on the Web, closes their browser, whatever—the row is left locked. For this reason you might want to shorten the active time-out.

lockRow() and unlock() methods

Related topics

To attempt a row lock, call the *lockRow*() method. It returns true or false to indicate success or failure.

Code an active lock in the *canEdit* event like this:

```
function query1_canEdit()
{
  var lRet = this.lockRow();  // Did you get the lock?
  if ( !lRet ) {              // If not
    //
    // Display a message or something to indicate failure
    //
  }
  return lRet;
}
```

A lock attempt during an *onNavigate* would be similar, except that there's no need to return the success value. You could code it more succinctly by putting the *lockRow*() call in the *if* statement:

```
function query1_onNavigate()
{
  if ( !this.lockRow() ) {    // If you didn't get the lock
    //
    // Display a message or something to indicate failure
    //
  }
}
```

In both cases, you need to release the lock by calling *unlock*(). For a lock that was used for editing, you would unlock when the editing is saved or abandoned. For a lock in *onNavigate*, you would release the lock during the *canNavigate*, when the user attempts to leave the row.

Indicating lock failures

Related topics

There are several ways to indicate a failed lock attempt. You could put an HTML text object on the form, which either always contains a status message of some sort, or is normally blank and is used to display alerts. If the lock fails, you can set the *text* property of that HTML object.

Or you could use a separate form (or page on the same form) that has the lock failure message and an OK button. Pressing the button closes the new form or takes the user back to the previous page.

Retrying lock attempts

Related topics

By default, a rowset's *lockRetryCount* property is set to zero, which means that if the *lockRow*() method fails, the lock is not retried and *lockRow*() returns false. This setting is suitable for active row locking, because failure to get a lock should mean that you cannot edit the row (because you've coded it that way). The user may then try to edit again.

With passive locks, set the *lockRetryCount* property to the number of people you expect might simultaneously attempt to post changes to any given row. This number probably wouldn't be any higher than 5, but you could play it safe and set the number artificially high, like 100. The idea to make sure that everyone gets their chance to post their changes. If the *lockRetryCount* is too low, then the passive lock will fail, an exception would occur, and their changes would be lost. Because passive locks are brief, you should never get a lot of simultaneous lock attempts, so users would never wait too long for the lock.

The *lockRetryInterval* interval determines the amount of time, in seconds, to wait between each lock attempt. This number should be low, typically 1 or 2, but it should allow enough time for another passive lock to finish its job and be released.

Other locking considerations

Related topics

Because of their opposite natures, you should not mix active and passive locks on the same table. Passive locks expect rows to be available if not immediately, while active locks hold for extended periods.

This is an important point not only for any single IntraBuilder application or multiple Web applications, but also with any applications that access the same tables. You might use an application on the LAN that accesses the same table you're accessing over the Web. The different applications must use compatible locking protocols—if not, your table will get corrupted. Also you must make sure they use the same types of locks.

Finally, if you're editing within a transaction, as discussed later, any locks—both active and passive—will not be released until the transaction is completed. This allows IntraBuilder to rollback the changes if requested.

Saving or abandoning changes

Related topics

Once changes have been made they can be saved or abandoned.

modified property

Related topics

Just because a rowset has been placed in Edit mode does not mean that changes have taken place. If there have been no changes, there's no point in trying to save the row.

The rowset's *modified* property is used to reflect whether changes have been made.

modified is set to true automatically whenever the *value* property of Field object is assigned a value, even if it's the same value that was already there. With *dataLink*ed components, the behavior depends on whether the form is run locally or remotely. When run locally, a change in the component, even if it was modified and changed back, is copied to the field when the component loses focus. If the component is visited but not changed (for example just tabbed through) *modified* is not affected.

When using a browser, IntraBuilder doesn't know when and if a change is made until the entire form is submitted. Then IntraBuilder compares the new values from the browser with those on the server. If the values are the same, then that field is considered not changed, even if it was modified but changed back in the browser. Any changed fields cause the rowset's *modified* property to be set to true.

IntraBuilder checks the *modified* property to see whether the rowset needs to be saved, even when you call the *save*() method. If *modified* is false, calling *save*() has no effect.

You can use this behavior to your advantage by manually setting *modified* to false to prevent automatic saving. This is especially useful when prefilling new rows, as discussed in Pre-filling the new row with default values.

Automatic saving

Related topics

Changes to a rowset are automatically saved when navigating off a row, closing the rowset, and in other situations. This means that if you're using *autoEdit*, you can make changes and navigate without ever having to explicitly save. This is more efficient, if less foolproof.

However, you get better control and more flexibility by using separate Edit and Browse modes. Therefore, when switching to Edit mode hide any navigation buttons to give the user only two clear choices: save or abandon.

To disable automatic saving during navigation, set the rowset's *modified* property to false in the *canNavigate* event.

Saving changes

Related topics

Call the rowset's *save*() method to save changes. The following sequence of events may occur:

IntraBuilder checks the rowset's *modified* property.

If it is false, nothing else happens.

If it is true, then the rowset's *canSave* event is fired. The usual job of *canSave* is to make sure the data is valid before letting it be posted. (See Validating data on a form.)

If there is an event handler assigned to the *canSave* event and that event handler returns false, the contents of the rowset are not saved.

If *canSave* returns true or there is no method assigned to the rowset's *canSave* event, then the actual save is attempted.

If the row is not already locked with an active lock, a passive lock is attempted. If the locks succeeds, then the data is posted and if there was a passive lock, it is released.

After the successful post, the rowset's *onSave* event fires. If *autoEdit* is false, the rowset goes from Edit mode back to Browse mode.

Abandoning changes

Related topics

The rowset's *abandon*() method can be used in many places. One of its main jobs is to abandon, or discard, any changes made to a rowset.

When *abandon*() is called, the *canAbandon* event fires. If it returns false, the abandon does not occur. There are few reasons why you wouldn't let someone abandon changes.

In abandoning the changes, the *value* properties of the fields take on their original values.

If *notifyControls* is true, then *dataLink*ed components are also refreshed. The rowset's *onAbandon* event fires.

If *autoEdit* is false, the rowset switches from Edit mode back to Browse mode.

Editing existing rows on a form

Related topics

The simplest way to enable editing on a form is to set the rowset's *autoEdit* property to true. Then any *dataLink*ed component may be changed, and any navigation will save the changes. All you would need is the *dataLink*ed components to display and edit the data, the navigation controls, a save button so that the user can save without navigating, and an abandon button.

To require a user to consciously switch to edit mode, you need an edit button. You can generate all these buttons by using the Form Expert. They simply call the corresponding method.

Using pages to display different modes

Related topics

If you're using separate Browse and Edit modes, you can use form pages to easily display a different set of buttons depending on the rowset's current mode.

Items on page 0 are displayed on every page. Place the *dataLink*ed controls and their corresponding HTML text labels there.

On page 1, place the Browse mode controls: navigation and the edit button.

On page 2, place the save and abandon buttons.

This clearly differentiates Browse and Edit modes, and makes the choices in Edit mode quite clear. The user can either save changes, abandon changes and go back to Edit mode, or go to another location in the browser in which case the edits are simply never committed and they eventually time out.

To switch to page 2 when editing, set the form's *pageno* property in the *onEdit* event handler:

```
function query1_onEdit()
{
  this.parent.parent.pageno = 2;  // Display the editing page
}
```

For a rowset event, *this* is the rowset. The rowset's *parent* is the query, and the query's *parent* is the form.

To go back to page 1 after a save or abandon, set the form's *pageno* property back to 1 in both the *onSave* and *onAbandon* event handlers.

Forms with more than one rowset

Related topics

Note that a form may contain more than one rowset (by having more than one query), and these rowsets may be in different modes. For example, you could have certain components *dataLink*ed to fields in a rowset in Browse mode and therefore not editable, and other components *dataLink*ed to fields in a rowset in Edit mode. If you have two rowsets and want to switch them to Edit mode, call *beginEdit*() for both; to save, call *save*() for both.

The TMD project: Editing existing rows

Related topics

Editing isn't a particularly useful option in a message database, but you can allow it. Ideally, you'll want to make sure that the person trying to edit a message is the original author. This of course would require that the users identify themselves when they access the database; you'll so this later in the TMD project.

For now, because there's nothing to prevent someone from creating a new message and putting whatever they want in the From field, support editing with passive locks.

Because the edit, save, and abandon buttons were all included when creating the form with the Form Expert, you need only place all the components on the right page and setup page switching.

Moving components to another page

Related topics

Now because the Form Expert places everything on page 1 by default, we need to move some components. The navigation, Add, and Edit buttons stay on page 1.

With VIEWER.JFM open in the Form Designer, move the following groups of components:

- Move the fields and labels to page 0 so that they appear on all pages.
- Move the Save and Abandon buttons to page 2.

To move the components to another page, repeat these general steps for each group:

1 Select all the components to be moved.

2 Press F11 to go to the Inspector without losing the selection on the group of components.

3 Click the Properties tab of the Inspector and expand the Visual Properties heading.

4 Set the *pageno* property to the desired value.

To move from page to page, press PgUp and PgDn while the main Form Designer window has focus or use the page buttons on the toolbar. The current page is displayed in the status bar.

Items on page 0 are displayed on all pages of a form.

Important Make sure to go back to page 1 before saving or running the form, because the page displayed when you save the form design will be the page that is first displayed when the form opens.

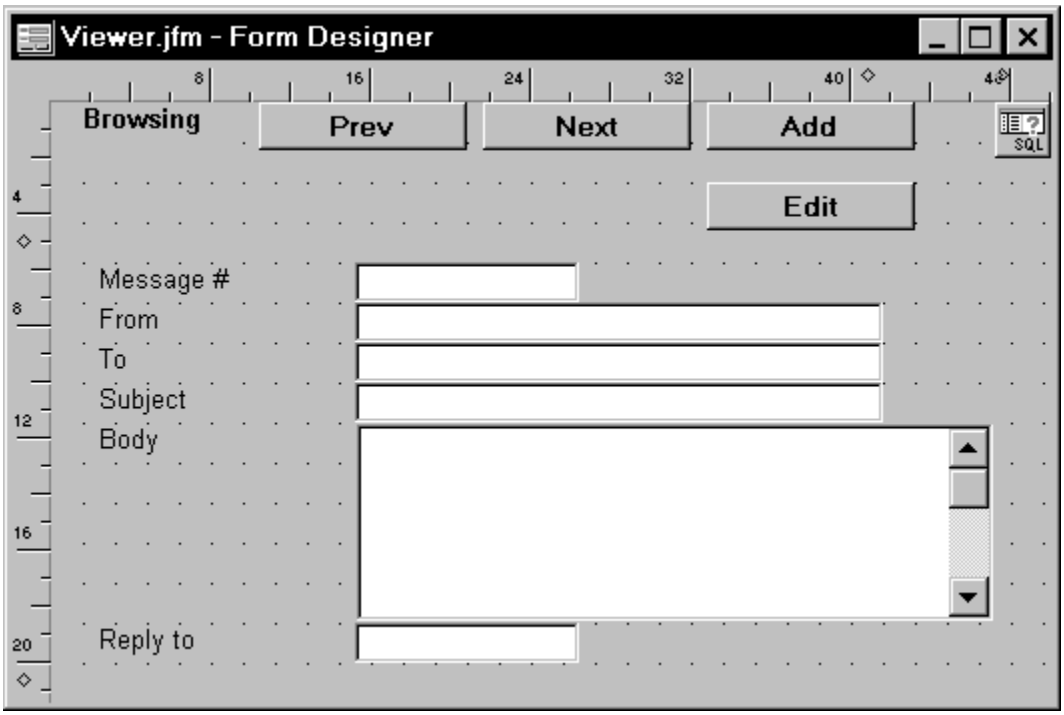
Rearranging components

Related topics

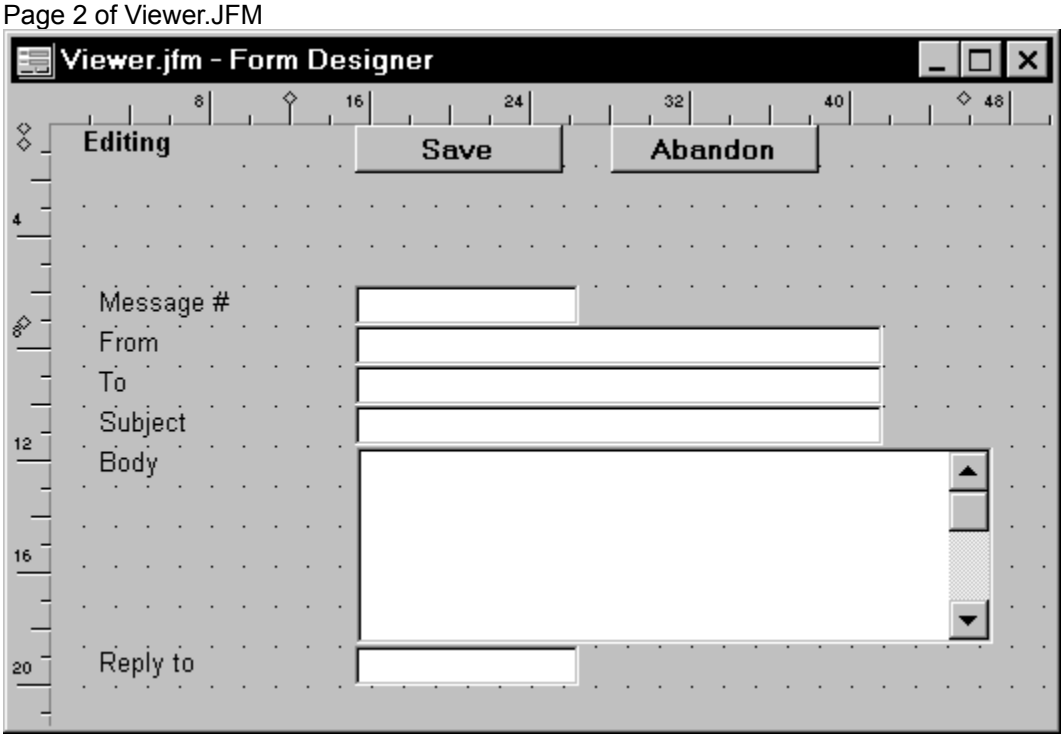
Delete the large title “Messages” and move the components up to fill the empty space. See Form page examples to view a suggested arrangement for the two form pages.

Form page examples

Page 1 of Viewer.JFM



Page 2 of Viewer.JFM



Creating page labels

Related topics

It would help to display a label or image on each page to indicate its purpose. For this part of the TMD project, simple HTML text labels will suffice.

Note that when you choose a component from the Component Palette, by default the component goes into the page currently displayed.

To create the page labels:

1. Switch to page 2.
2. Place a HTML component in the upper left corner of the form and make sure it is selected.
3. In the Inspector, change its *text* property to “Editing.”
4. Change its *name* property to editLabel.
5. Switch to page 1 and do the same thing, with the *text* property “Browsing” and the *name* property browseLabel.

See Form page examples to see how these pages should look.

Using common event handlers

Related topics

Type the following code for the rowset's *onEdit* event handler to switch to page 2:

```
function messages1_onEdit()
{
  this.parent.parent.pageno = 2;  // Display the editing page
}
```

You can code separate event handlers for *onSave* and *onAbandon* to switch back to page 1. Or you can code a single event handler and link that to multiple events.

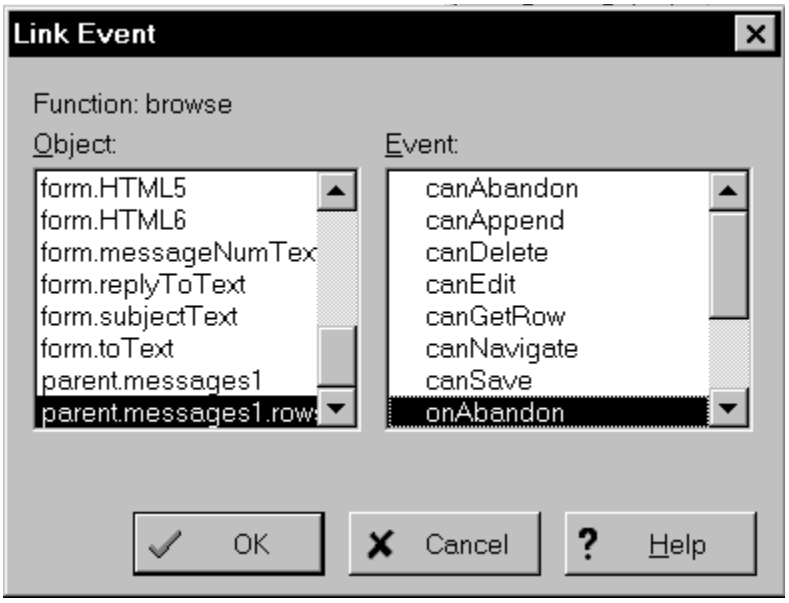
To use common event handlers:

1. Code the *onSave* handler to switch back to page 1:

   ```
   function messages1_onSave()
   {
     this.parent.parent.pageno = 1;  // Display the browsing page
   }
   ```

2. When that's done, right-click the Method Editor with the event handler displayed and choose Link Event.
3. In the Link Event dialog box, choose the query's rowset object in the Object list and the *onAbandon* event in the Event list.
4. Because the event is shared by both the *onSave* and *onAbandon* events, edit the method in the Method Editor to change its name to "browse." The event should now look like this:

   ```
   function browse()
   {
     this.parent.parent.pageno = 1;  // Display the browsing page
   }
   ```



Validating data

Related topics

There's nothing worse in a database application than bad data. It can cause a myriad of problems. Therefore, do everything you can to make sure that the data saved in the table is valid. For example, fields that should not be left blank should be filled in.

When data has been changed, IntraBuilder always tries to save it. The key to data validation is the *canSave* event.

canSave event

Related topics

The rowset's *canSave* event fires when the following happens:

1 An action causes IntraBuilder to check if data has been changed. This is either:

- The *save*() method is explicitly called; or
- One of the following actions causes an implicit save check:

Action	Code	can- event	on- event
Navigation	*next*(), *first*(), *last*(), *goto*()	*canNavigate*	*onNavigate*
Deactivating the query	*parent.active = false*	*canClose*	*onClose*
Appending a new row	*beginAppend*()	*canAppend*	*onAppend*
Switching to Filter mode	*beginFilter*()	none	none
Switching to Locate mode	*beginLocate*()	none	none

2 If an action has its own can- event, it is fired first. If that event returns false, then nothing else happens, because the entire action has been prevented.

3 If the can- event returns true, or if there is no separate event, then the rowset's *modified* property is checked.

4 If *modified* is false, then no save attempt occurs.

If it is an implicit save check, the initiating action occurs. If it is an explicit *save*() call, it has no effect.

5 If *modified* is true, then the rowset's *canSave* event fires.

6 If *canSave* returns false, then the row is not saved.

If it is an implicit save check, the initiating action does not occur.

7 If *canSave* returns true, or if there is no *canSave* event handler, the row is saved.

8 If the row is saved, the *onSave* event fires afterward.

- If it is an implicit save, then the initiating action finally occurs after the *onSave*.
- Then its own on- event, if any, fires.

For example, if a user changes some data and attempts to navigate to the next row, the *canNavigate* event fires.

Typically, the *canNavigate* event handler, if any, does not attempt to validate the data. It leaves that up to *canSave*. When the *canSave* checks the data that has been changed, if the data is invalid the navigation attempt is prevented, as if *canNavigate* had returned false instead of true. *canNavigate* must have returned true; otherwise the *canSave* would never have fired.

Neither event generates an error or displays a message unless you do so yourself. When the navigation is prevented, it would be as if pressing the navigation button had no effect. If you do nothing else, the invalid data would still be displayed, which gives the user the opportunity to correct the data.

Once *canSave* returns false, the user has three options:

- Correct the data and try again.
- Abandon the changes, which causes the following:
- the row reverts to its previous values
- the *modified* flag is cleared (set back to false), which
- prevents *canSave* from being fired
- Ignore the problem entirely and go to another page through the browser. This could even be another page in your Web application. In any case, the form with the invalid data will eventually time out and the changes will be discarded.

You should indicate to the user that the navigation (or whatever the initiating action was) failed because the data was invalid. There are a number of ways to do this, and you can go so far as to visually indicate the fields that have problems, as discussed in Field-level validation.

You can eliminate all implicit save failures by using separate Edit and Browse modes and especially a

separate edit page on your form. Then *canSave* is fired only when you explicitly call the *save*() method. And *save*() has the advantage of being the only method that returns *canSave*'s return value, so you can act accordingly.

With implicit saves, the initiating methods return values with their own meanings. Make sure your code does not assume the save was successful when it might fail.

The *canSave* event is unique in its ability to prevent other actions. Because of this and because it's always fired when saving, you need only write your row validation code once in the *canSave* event to cover all contingencies.

Row-level validation

Related topics

In the *canSave* event handler, check the *value* properties of the fields to make sure they are valid.

If there is more than one field or validation rule to check, create a result variable at the beginning of the *canSave* event handler. Initially set it to true and then set it to false in every check that fails.

For example, with the TMD project, the To, From, and Subject fields may not be blank:

```
function messages1_canSave()
{
  var lRet = true;    // Logical return value defaults to true; assume
everything is OK
  var cErrors = "";   // Text to contain errors
  if ( this.fields[ "From" ].value == null ) {
    lRet = false;
    cErrors += "- <B>From</B> field cannot be blank<BR>";
  }
  if ( this.fields[ "To" ].value == null ) {
    lRet = false;
    cErrors += "- <B>To</B> field cannot be blank<BR>";
  }
  if ( this.fields[ "Subject" ].value == null ) {
    lRet = false;
    cErrors += "- <B>Subject</B> field cannot be blank<BR>";
  }
  if ( !lRet ) {  // If there are problems set the error message
    this.parent.parent.statusLabel.text = "The data cannot be saved
because<BR>" + cErrors;
  }
  this.parent.parent.statusLabel.visible = !lRet;  // and display it
  return lRet;    // Return success or failure
}
```

By putting each check in its own *if* statement, you can build a message with details on the specific errors or even visually indicate which fields have problems. This *canSave* event handler displays the message in an HTML component named statusLabel (that has not been created yet). When there is no error, the component is invisible.

Field-level validation

Related topics

Field objects have a *canChange* event which determines whether their *value* properties can be changed. This is usually the result of attempting to echo the values in the form components to their *dataLink*ed fields in the rowset when the form is submitted. The *canChange* event also fires when directly assigning a value to the field's *value* property.

If the *canChange* event returns false, the new value is not written to the field. This does not affect *canSave*, except that if no fields are written to, the rowset's *modified* property is never set to true, so the *canSave* will not fire and the row will not be posted to the table. This can lead to an undesirable effect: if the user places invalid data in all the fields, then the edits are simply lost. It would be better to be informed that the data is invalid and needs to be corrected.

You can also use *canChange* for field morphing, that is, for displaying a different value than that stored in the table. For example, you can store a customer number in a field but display a customer name in the form component.

When writing to a morphed field, the *canChange* event handler changes the *value* of the field, but returns false so that the value in the component is not written to the field. This does not mean that the data is invalid. Also, because the *value* property is assigned inside the *canChange* event handler, the rowset's *modified* property is set to true. Field morphing is discussed in more detail in JavaScript forms

All the field's *canChange* events, if any, fire before any other actions. When run locally, changed component values are echoed when the component loses focus. When run remotely from a browser, all the changed components are echoed when the entire form is submitted. This must occur before attempting the actual reason for the submission, like navigation, so that IntraBuilder can accurately determine if values have been changed and need to be saved.

Client-side validation

Related topics

In addition to using *canSave* on the server, you can also do client-side validation (provided that the user's browser supports JavaScript). Client-side validation should always augment but never replace server-side validation. Even if you can guarantee that all your users will use a browser that supports JavaScript, they might disable it. So you would always want to make sure that the data is valid on the server before posting it.

Client-side JavaScript is discussed in Client-side JavaScript.

Validating data on a form

Related topics

Because the *canSave* event is fired whenever data is about to be saved and needs to be validated, you need only code a single *canSave* event handler as shown in Row-level validation. If you have more than one rowset on the form, you will need to make sure there is a *canSave* event handler for each rowset.

The TMD project: Validating data

Related topics

In the TMD project, the To, From, and Subject fields should not be blank.

If you haven't done so already, set the rowset's *canSave* event to the code for row-level validation (as shown in Field-level validation).

That code used an HTML text object named statusLabel, so you'll need to create it:

1. Switch to the Edit mode page (page 2) in the Form Designer.
2. Create an HTML component and place it just below the Editing label.
3. Stretch the HTML component so that it extends across the width of the form.
4. Set its *name* property to statusLabel.
5. Set its *visible* property (under the Access Properties heading) to false.
6. It starts out invisible, and is made visible if there is an error. To make sure it's invisible every time the user starts editing, set its *visible* property to false after saving or abandoning changes. Add the highlighted line to the browse() method:

```
function browse()
{
  this.parent.parent.statusLabel.visible = false;
  this.parent.parent.pageno = 1;  // Display the browsing page
}
```

Adding new rows

Related topics

Most database applications allow the user to add new rows. In IntraBuilder new rows are added through Append mode.

Append mode

Related topics

A rowset's fourth mode—in addition to Close, Browse, and Edit—is Append. Calling the *beginAppend*( ) method attempts to put the rowset in Append mode. The rowset's *canAppend* event is fired, and if the current contents of the row need to be saved, the usual *canSave* event interaction occurs. *beginAppend*() returns true or false to indicate whether the switch was successful or failed for any reason, such as *canSave* returning false.

If the rowset successfully switches to Append mode, then the row buffer is blanked, which in turn blanks *dataLink*ed components. After the row buffer is blanked, the rowset's *onAppend* event fires.

At this point, Append mode is similar to Edit mode, in that the idea is to make changes and save them as a new row or abandon the mode, both of which cause the rowset to go back to Browse or Edit mode (depending on the rowset's *autoEdit* property). One difference is that there is no active or passive row-locking, because it's a new row.

When the new row is saved, either through an explicit *save*() call, navigation (which would be relative to the new row), or another *beginAppend*(), the row validation mechanics with the *modified* property and *canSave* event occur. If the save proceeds, then the entire set is momentarily locked to add the new row.

Append mode is used for both batch data processing in scripts and interactive data entry. With batch data processing, after switching to Append mode, the *value* properties of the fields are set and the row is saved with *save*(). In interactive data entry, the user types values into components on a form. Usually, these components are directly *dataLink*ed to fields so that when the form is submitted, the values in the components are echoed in the fields.

There are also some hybrid approaches. You could take the values in non-*dataLink*ed components submitted on the form, and then manually set the *value* properties of the fields based on calculations or lookups. Or you could pre-fill the new row with default values before users see the form, which they can then accept or change.

Pre-filling the new row with default values

Related topics

To pre-fill a new row, set the fields' *value* properties after switching to Append mode. The rowset's *onAppend* event is the easiest place for this. For example, you can set default quantities for orders, or default values for the most-likely area codes for phone numbers or for most-likely states for addresses, and so on.

Because you have assigned values to the *value* properties, the rowset's *modified* property will automatically be set to true. In most cases, you should set the *modified* property back to false at the end of the *onAppend* event handler. This indicates that the row has not been modified by the user. If the user does nothing else and submits the form as-is, it will not be saved.

The exception is if the default values you set are acceptable as data entry. This is rare, however.

Although the rowset's *onAppend* event is the natural place to pre-fill the new row, it would work only if you add only one type of row. For example, in the TMD project, you can either add a completely new message or compose a reply to an existing message. In both cases, you would want to assign the user's name to the From field (assuming you knew what it was—and in the current version you don't), so that could go in the *onAppend* event. But in the reply only, you would also need to fill in the Reply To field with the appropriate message number, copy the name from the From field in the original message to the To field in the reply, and duplicate the Subject.

Because you would have two separate buttons, one for a new message and one for a reply, the new message button's *onServerClick* event handler would just call *beginAppend*(), but the Reply button's *onServerClick* would also set the Reply To field.

To illustrate this theoretical situation, consider this code (but don't enter it):

```
function newButton_onServerClick()
{
  this.form.rowset.beginAppend();
}
function replyButton_onServerClick()
{
  var nReplyTo = this.form.rowset.fields[ "Message #" ].value;  // Get the
message #
  var cTo      = this.form.rowset.fields[ "From"      ].value;  //   name
  var cSubject = this.form.rowset.fields[ "Subject"   ].value;  //   and
subject first
  if ( this.form.rowset.beginAppend() ) {                       // Switch to
Append mode
    this.form.rowset.fields[ "Reply to" ].value = nReplyTo;     // Fill in
the fields
    this.form.rowset.fields[ "To"       ].value = cTo;
    this.form.rowset.fields[ "Subject"  ].value = cSubject;
    this.form.rowset.modified = false;                          // Clear
modified flag
  }
}
function messages1_onAppend()
{
  this.fields[ "From" ] = this.parent.parent.userName;          // Fill in
user's name
  this.modified = false;                                        // Clear
modified flag
}
```

Note that the *modified* property is set to false in both the methods where a *value* is assigned, because the *onAppend* will fire right after the *beginAppend*() completes successfully.

Also, because the reply button is a component on the form, getting to the rowset's *modified* property uses a different object reference than the rowset's *onAppend* handler.

Finally, the code to pre-fill the Reply To field executes only if the row successfully switches to Append mode.

Adding rows to an empty table

Related topics

When you have a finished application but the table is initially empty, the form displays the dreaded phantom row—which is nothing. Because there are no rows, you can't navigate, search, or edit. To do anything, you'll first need to put some data in the table:

1. Create a separate empty table page to display the empty *dataLink*ed components on page 0.
2. Create a message on the page informing the user that the table is empty.
3. Create an Add button to switch to Append mode.

You can detect whether there is data in the table by checking the rowset's *endOfSet* property when the form first loads in the form's *onServerLoad* event. Also check in the rowset's *onAbandon* event, in case the user starts with an empty table, tries to add a row, and abandons it.

Adding new rows in a form

Related topics

When using *dataLink*ed components to display data on a form, you can easily enable users to add new rows by including an Add button. The Form Expert does that for you or you can create one yourself. The Add button calls the rowset's *beginAppend*() method to put the rowset in Append mode.

To save or abandon the new rows you need Save and Abandon buttons, which you should already have for editing.

If the form has more than one rowset, you may have one Add button that calls *beginAppend*() for all the rowsets, or you may have separate Add buttons for each rowset. You also need to call *save*() or *abandon*() for each rowset.

The TMD project: Adding new rows

Related topics

In the TMD project, users can add two kinds of messages: replies to existing messages and brand new messages to start a thread. In this version of TMD, new messages are simple new rows. Replies take the existing message number and store them in the Reply To field, and copy the From field in the original message into the To field in the reply.

To implement these changes:

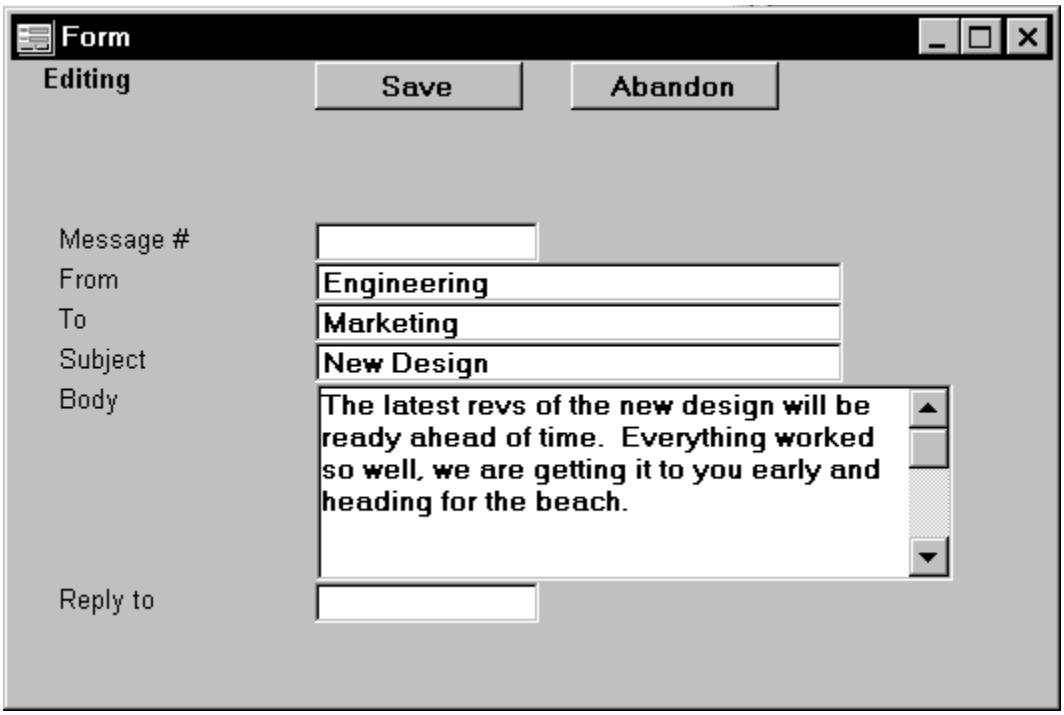
1. Change the *text* property on the Add button to read “New” for new messages.
2. Change the Add button’s *name* to newButton.
3. On the same page (page 1), drag another button from the Component Palette.
4. Set this button’s *name* to replyButton.
5. Set its *text* to “Reply.”
6. Type in this code for the Reply button’s *onServerClick* event handler:

```
function replyButton_onServerClick()
{
  var nReplyTo = this.form.rowset.fields[ "Message #" ].value;
  var cTo      = this.form.rowset.fields[ "From"      ].value;
  var cSubject = this.form.rowset.fields[ "Subject"   ].value;
  this.form.rowset.beginAppend() )
  this.form.rowset.fields[ "Reply to" ].value = nReplyTo;
  this.form.rowset.fields[ "To"       ].value = cTo;
  this.form.rowset.fields[ "Subject"  ].value = cSubject;
  this.form.rowset.modified = false;                          // Clear
modified flag
  this.form.rowset.refreshControls();                         // Update
controls on form
}
```

7. Because this version of the TMD project doesn’t know the user’s name, you cannot fill that in automatically in the *onAppend* event as shown in the pre-fill code. However, you can switch to the editing page (page 2) in the *onAppend* event handler, just as you did in the *onEdit* event handler discussed earlier:

```
function messages1_onAppend()
{
  this.parent.parent.pageno = 2;  // Display the editing page
}
```

Now try out the new form. Switch to Run mode. Try entering a few new messages and a few replies. Don’t type anything into the Message # and Reply to fields; they are filled in automatically.


Form
Editing
Save
Abandon
Message #
From
Engineering
To
Marketing
Subject
New Design
Body
The latest revs of the new design will be ready ahead of time. Everything worked so well, we are getting it to you early and heading for the beach.
Reply to

Filtering rows

Related topics

IntraBuilder supports a Filter mode that makes it easy to implement a Filter-By-Form feature, to filter out rows and display only those that match specified conditions.

Filter mode

Related topics

The idea behind Filter mode is to assign values to the field objects of the fields you want to match. Fields you don't care about should not be touched.

The individual field conditions are additive; that is, they are joined by a logical and. For example, if you set the To field object's *value* to "Borg" and the Subject field object's *value* to "Sleep" then IntraBuilder will show only those messages addressed to "Borg" with subject "Sleep."

You can set the *value* properties of the fields through code or let the user type values into form components. These values are then echoed into the fields as usual.

To start Filter mode, call the rowset's *beginFilter*() method. IntraBuilder attempts to put the rowset in Filter mode, with the *modified*/*canSave* interaction first. There are no canFilter or onFilter events. *beginFilter*() returns true or false to indicate whether the change to Filter mode was successful or failed for any reason, such as *canSave* returning false.

If the rowset successfully switches to Filter mode, then the row buffer is blanked, which in turn blanks *dataLink*ed components.

To cancel Filter mode, call *abandon*(). The contents of the previously displayed row are displayed.

To set the filter, call the rowset's *applyFilter*() method.

- If there are rows that match the filter conditions, the row cursor is positioned to the first matching row, which fires the *onNavigate* event, and *applyFilter*() returns true.
- If there are no matching rows, the row cursor is positioned at the end-of-set and *applyFilter*() returns false.

Whether you call *abandon*() or *applyFilter*() or whether the filter finds a match, the rowset reverts to Browse or Edit mode, depending on the rowset's *autoEdit* property.

The filter is active until it is cleared by calling the *clearFilter*() method. Since whatever row you were on must have matched the filter condition, no navigation occurs when you clear the filter.

filterOptions property

Related topics

The *filterOptions* property is an enumerated property that controls how the *value* properties in the field objects are matched against the values in the table. The options are:

Value	Effect
0	Match length and case
1	Match partial length
2	Ignore case
3	Match partial length and ignore case

Filter constraints

Related topics

If a row is edited so that it no longer matches an active filter condition, when it is saved it temporarily drops out of the rowset and the row cursor is moved to the next matching row. This row cursor movement causes an *onNavigate* event to fire.

If there is no matching row after the just-edited one, the row cursor moves to the end-of-set.

Filtering rows on a form

Related topics

To implement Filter-By-Form you need only:

- the *dataLink*ed components you're using to display table data
- a button to call the *beginFilter*() method
- and a button to call *applyFilter*()

By the way, you can quickly add a pre-built Filter-By-Form button by using the Form Expert.

Create a separate filter page with brief instructions on how the filtering works and a Select object to choose the type of matching. This page could contain the *applyFilter*() button and a button to *abandon*(), just in case.

Check the return value from *applyFilter*(). If it returns false, there were no matches. Display a message to inform the user that there were no matches and return to Filter mode so the user can try another filter or abandon.

To indicate that a filter is active and give the user a direct way of turning it off, put a button on the main browsing page that calls *clearFilter*(). You can hide this button most of the time by setting its *visible* property to false when designing the form and then setting it to true only when *applyFilter*() returns true.

The TMD project does not currently support filtering. Please refer to the latest online documentation and samples from Borland Online.

Locating rows

Related topics

Basic locating is almost identical to filtering, except that the condition is not persistent. It either finds a match or not.

Locate-By-Form may sometimes be referred to as “Query-By-Form.”

Locate mode

Related topics

The idea behind Locate mode is to assign values to the field objects of the fields you want to match. Fields you don't care about should not be touched.

The individual field conditions are additive; that is, they are joined by a logical and. For example, if you set the To field object's *value* to "Locutus" and the Subject field object's *value* to "Assimilation" then IntraBuilder will try to find the first message addressed to "Locutus" with subject "Assimilation."

You can set the *value* properties of the fields through code or let the user type values into form components. These values are then echoed into the fields as usual.

To start Locate mode, call the rowset's *beginLocate*() method, which attempts to put the rowset in Locate mode, with the *modified*/*canSave* interaction first. (There are no canLocate or onLocate events.) *beginLocate*() returns true or false to indicate whether the switch to Locate mode was successful or failed for any reason, such as *canSave* returning false.

If the rowset successfully switches to Locate mode, then the row buffer is blanked, which in turn blanks *dataLink*ed components.

To cancel Locate mode, call *abandon*(). The contents of the previously displayed row are displayed.

To execute the search, call the rowset's *applyLocate*() method.

- If there's a match, the row cursor is positioned to the first matching row, which fires the *onNavigate* event, and *applyLocate*() returns true.
- If there are no matching rows, the row cursor is positioned at the end-of-set and *applyLocate*() returns false.

Whether you call *abandon*() or *applyLocate*() or whether a match is found, the rowset reverts to Browse or Edit mode, depending on the rowset's *autoEdit* property.

Locating other matching rows

Related topics

Once you successfully *applyLocate*(), you can call the *locateNext*() method to find other rows that match. This differs from using a filter, which is always in effect until it is cleared. With a filter, basic navigation like *next*() and *first*() will go only to those rows that match. With locating, basic navigation works as usual, and *locateNext*() takes the user to a matching row, relative to the current position.

locateNext() takes a single numeric parameter, just like *next*(). It indicates how many rows to move and in which direction, positive or negative.

- If no number is specified, the default is 1, which means the next matching row.
- The previous matching row would require a parameter of –1.
- If there are no more matching rows in either direction, *locateNext*() stops at the end-of-set and returns false.
- If *locateNext*() finds the desired match, it returns true.

locateOptions property

Related topics

The *locateOptions* property is an enumerated property that controls how the *value* properties in the field objects are matched against the values in the table. The options are identical to the *filterOptions* property:

Value	Effect
0	Match length and case
1	Match partial length
2	Ignore case
3	Match partial length and ignore case

Locating rows on a form

Related topics

To implement Locate-By-Form, use:

- the *dataLink*ed components you're using to display table data
- a button that calls the *beginLocate*() method, and
- a button that calls *applyLocate*()

By the way, you can quickly add a pre-built Locate-By-Form button by using the Form Expert.

Create a separate locate page with brief instructions on how the searching feature works. Add a Select object to choose the type of matching. This page also contains the *applyLocate*() button and a button to *abandon*(), just in case.

Check the return value from *applyLocate*(). If it returns false, there were no matches. You can display a message to inform the user that there were no matches and return to Locate mode so that the user can try another filter or abandon.

Once a match has been found, you have the option of using separate Next Match and Previous Match buttons in addition to the basic Next and Previous navigation buttons. You could place these buttons on the main browsing page and keep them invisible most of the time by setting their *visible* properties to false.

The TMD project does not currently support locating. Please refer to the latest online documentation and samples from Borland Online.

Deleting rows

Related topics

Not all applications allow users to delete data. This is particularly a consideration when giving access over the Web.

The deletion process

Related topics

To delete the current row, call *delete*(). Unlike most other rowset methods, the rowset's *modified* property is not checked and *canSave* is never fired, because the row is to be deleted.

The *canDelete* event is fired.

- If it returns false, the delete does not occur.
- If it returns true, or there is no *canDelete* event handler:
- The current row is deleted.
- The current row immediately drops out of the rowset and cannot be recovered.
- The row cursor is moved to the next available row, or the end-of-set if there are no more rows.
- Then the *onDelete* event fires.

Deleting the last row

Related topics

You can try to detect if there no more rows in the table in the *onDelete* event. If there are no more rows, switch to an empty table page that gives the user the sole option of adding a new row.

Check if a filter is active, because that would delete any non-matching rows.

Deleting rows on a form

Related topics

The Form Expert gives you the option of including a Delete button, which simply calls the *delete*() method.

The TMD does not allow message deletion by the user.

Project summary

Related topics

At this point, the Threaded Message Database project supports basic functionality. Most of the capabilities were set up by using the Form Expert. Then you added a few cosmetic rearrangements and events handlers.

While it adequately serves to demonstrate basic database concepts, the TMD project falls far short in the real world in a number of areas. In the next series of topics, the TMD project will be greatly enhanced. In the process, many real-world application techniques and considerations are examined.

Customizing the application

Related topics

While the previous series of topics discussed basic database concepts and how they are applied in IntraBuilder, this series approaches application development from the opposite perspective: using IntraBuilder techniques to achieve specific application goals.

Application design is about making things work the way you want, within the limits that confront you. Specifically, database applications over the Web have particular constraints and advantages that you don't find in LAN-based applications.

The features described in this series greatly enhance the simple Threaded Message Database project discussed throughout the previous series. In the process of adding these features, powerful IntraBuilder techniques are demonstrated. You will learn:

- How to build a custom login form
- Linking objects
- Linking forms
- Linking events
- Passing parameters
- Field morphing

Other features and techniques are detailed in documents on the Borland IntraBuilder Web site and discussed on the IntraBuilder support forum.

TMD project features

Related topics

By the end of the previous series of topics, you had completed a basic, functional Threaded Message Database application with the following features:

- An autoincrementing message number, so that each new message is automatically assigned a unique number.
- Fields for the sender, recipient, message subject, and message body.
- Forward and backward navigation through messages.
- When replying to a message, TMD automatically copies the From field from the original into the To field of the reply and duplicates the subject. But TMD does not fill in your name because it doesn't know who you are.
- TMD stores the number of the message you're replying to, but there is no direct way to go to that message.

While this feature set has served to illustrate basic database concepts, it falls well short of a usable application. To remedy that, we will now implement the following features:

- Users are required to login. This not only ensures against impersonation, but because the TMD application now knows who you are, it can fill in your name automatically when needed.
- After composing a reply or new message, users are returned to the message they were responding to, rather than leaving them with the reply they just wrote.
- Users can always display a reply's parent message (that is, the message to which the reply message is responding). Users are free to navigate in the thread and resume where they left off.
- Different message sections are supported, so that user can choose a relevant topic or grouping of messages to view. Sections could be used to classify messages any way you want.
- Messages are sorted in thread order, that is, grouped by subject.
- The TMD tracks the user's High Message Number (HMN), the highest numbered message this particular user has read. Whenever the user logs in, he or she skips all older messages. (The user can choose to reset his or her HMN to a lower number.)

As we introduce each new feature in the message browser, each IntraBuilder feature needed to support it is discussed in the context of how it applies to the TMD application and how it works in general. Then a procedure is given that you can follow to implement the new TMD feature.

Note that the specific IntraBuilder features used to implement a feature in the application are not necessarily the only ones that will work. Whenever possible, alternatives are discussed, and more importantly, seeing how a feature works in context may give you ideas when you consider your own application design.

Table and field modifications

Related topics

In the process of designing and refining a database application, it’s not unusual to change the structures of the tables in your database. You may add some fields, change their sizes, and occasionally replace some fields.

To support the new features in the message database, we will modify the message table, and add two new tables, one for users and one for sections.

Modifying an existing table

Related topics

To modify an existing table, no one else may have it open. This means the table may not be used in any active queries anywhere, including queries in forms, reports, and queries that you create in the Script Pad. The table may be referred to in the *sql* property of a query, but in that case, the query must be closed; its *active* property must be false.

Different table formats have slightly different rules regarding what you can safely do, but in general it's safe to make fields bigger. You can make character fields longer and store more digits for numbers. On the other hand, making fields smaller might cause data loss: character strings might get truncated, and numbers might overflow, losing their values.

If you don't care about the data in the table, especially during the early developmental stages, you could empty the table first before changing its structure. Otherwise IntraBuilder will waste its time making backups to copy the old data into the new table.

A Database object's *emptyTable*() method will quickly remove all the rows in a table, without double-checking for confirmation. To empty a table, no one else may have it open. For example, to empty the MESSAGES.DB table, use the default database, which handles Standard (DBF and DB) tables in the current directory:

```
_sys.databases[0].emptyTable( "MESSAGES.DB" )
```

You can also modify tables by using the SQL ALTER TABLE command, through a Database object's *executeSQL*() method. For example, to remove the field Weight from the MEMBERS.DB table, you can call *executeSQL* through the default database:

```
_sys.databases[0].executeSQL( "alter table MEMBERS.DB drop column WEIGHT" )
```

You can use that inside an application, but for interactive work, the Table Designer is friendlier.

The TMD project: Modifying tables

Related topics

You need to modify the Messages table and create the Users and Sections tables. The fields are not explained here, but rather as they are used later in this series.

Messages table

Related topics

To modify the Messages table,

1. Empty the MESSAGES.DB table by typing the *emptyTable*() statement in the Script Pad:

```
_sys.databases[0].emptyTable( "MESSAGES.DB" )
```

2. Double-right-click the MESSAGES.DB table in the Explorer to modify it.
3. Change the From field from an Alpha to Long, and add # of replies, Thread root, Posted, and Section. The table structure should now be:

Name	Type	Width
Message #	AutoIncrement	4
From	Long	4
To	Alpha	30
Subject	Alpha	30
Body	Memo	10
Reply to	Long	4
# of replies	Long	4
Thread root	Long	4
Posted	Timestamp	8
Section	Long	4

4. If the Inspector is not visible next to the Table Designer window, press F11 or choose View|Inspector from the menu. Click the Posted field in the Table Designer window. In the Inspector, set the *default* property of the Posted field to “NOW.” This writes the local system date and time of the IntraBuilder Server.
5. Save the table and close the Table Designer.

Users table

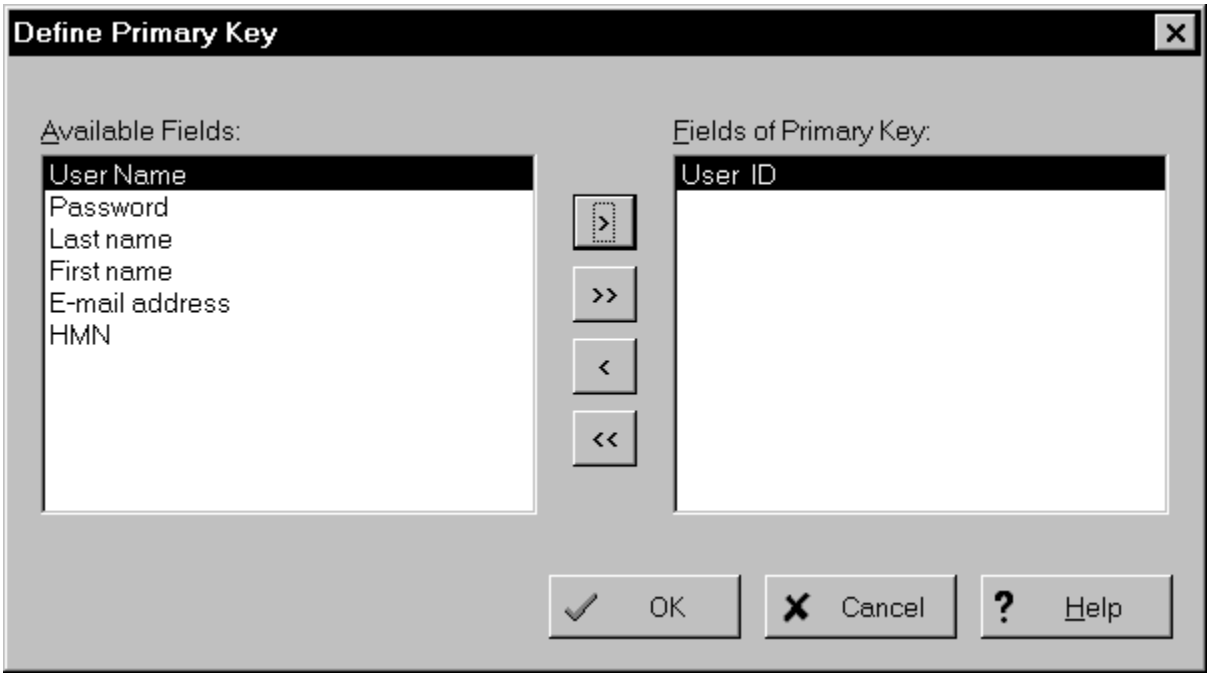
Related topics

To create the Users table,

1. Double-click the (Untitled) icon in the Tables tab of the Explorer.
2. Choose Designer. Make sure the table type is Paradox.
3. Create the following fields:

Name	Type	Width
User ID	AutoIncrement	4
User name	Alpha	30
Password	Alpha	30
Last name	Alpha	20
First name	Alpha	20
E-mail address	Alpha	60
HMN	Long	4

4. In the field Inspector, set the *default* property of the HMN field to 0.
5. In the Structure menu, choose Define Primary Key.
6. Select the User ID field as the only primary key field and click the right arrow to add it to the list of primary keys. Then click OK.
7. Save the table as USERS.DB and close the Table Designer.



Sections table

Related topics

Another way to get to the Table Designer is directly from the Script Pad:

1. In the Script Pad, type:

   ```
   _sys.tables.design()
   ```

2. Make sure the table type is Paradox.
3. Create the following fields:

Name	Type	Width
Section #	Long	4
Name	Alpha	25

4. In the Structure menu, choose Define Primary Key.
5. Select the Section # field as the only primary key field, click the right arrow to add it to the list of primary key fields, and click OK.
6. Save the table as SECTIONS.DB and close the Table Designer.

The TMD login form

Related topics

Users will now be required to login to the TMD application. This will be a separate form, the first form they will see directly from their browsers instead of the VIEWER.JFM form.

The login form has

- a Text object for the login name,
- a Password object for the password,
- a Button object to try to login, and
- a Button object for new users.

Registered users will type their chosen user name and password to go directly to the TMD Viewer form. For new users:

1. The user clicks the New User button.
2. The login form displays its new user registration page. Here the user can type her full name and E-mail address.
3. The user clicks the Submit button.
4. The TMD application suggests a login name.
5. The user can then set her login name and password. A user cannot choose a login name currently used by someone else.
6. When the user successfully sets her login name and password, she is automatically logged in, and the TMD Viewer form is displayed.

Multiple queries in a form

Related topics

New user information is added to the Users table by using the rowset's Append mode .

In the middle of the append, the TMD application must do a search to see if the new user's login name is already used by someone else. You cannot do this search with the primary rowset because it's in Append mode, so you'll need to open the Users table in another query, that is, you'll need two queries on the same form. (Remember that every query has one rowset, and every rowset is part of a query.)

Usually you add multiple queries to a form to access different tables. (In fact, later in this series the Viewer form will use multiple tables.) But you can also open the same table in different queries. Each query has its own rowset, and each rowset has its own *state* property (which indicates the mode), its own row cursor, and so on. You can therefore add a row in one rowset while doing a lookup in the same table in another rowset.

Implementing multiple queries on a form is simple: Just drag one from the Component Palette, make it active, and set its *sql* property. An easier way is to drag a table from the Explorer—IntraBuilder's or the Windows Explorer—onto the form surface, which sets the *sql* property for you and makes the query active.

Queries on a form need not always be active. You activate a query by setting its *active* property to true. While a query is active, it consumes a small amount of memory and resources. In a case like the login form, where the second query is used only for new users, you can leave the query deactivated until needed.

When adding a query, make sure it is attached to the correct Database and Session object. By default, a query is attached to the default database in the default session, which is designed to access Standard (DBF and DB) tables in the current directory. This is sufficient for the login form.

If there is a Session or Database object on the form when a Query object is added, the Query object is automatically attached to the Session and Database objects by setting its *session* or *database* property (or both) to the object on the form. If there are multiple Session or Database objects on a form (this is rare), you'll need to make sure that the Query object is attached to the correct Session and Database objects.

Query naming

Related topics

When you drag from the Component Palette, the first Query object is named query1, the second query2, and so on. While a single query's name isn't that important, especially because its rowset is assigned to the form's *rowset* property, referring to multiple queries by number can be confusing.

When you drag an existing table to a form, the Query object created automatically generates a name based on the table name, such as *messages1* for the first Query object created by using the MESSAGES.DB table.

Because Query objects do not have a *name* property, you cannot edit their names inside the Form Designer. However, you can change the name in the JFM source code. Simply edit the form as a script and find the *with* statement where the query is created (where the name appears). A reference to the Query object is assigned as a property of the form and the name of that property is used as the name of the query. (In the form's constructor, the form is referred to by the reference *this*.)

As a hypothetical illustration, to change the default name of a Query object from *query2* to *qUsers*, you would change the first line of

```
with (this.query2 = new Query()){
  sql = "select * from USERS.DB";
  left = 50;
  top = 0;
}
```

to

```
with (this.qUsers = new Query()){
```

and save the script.

The TMD project: Creating the login form

Related topics

(Click here for illustrations of finished forms.)

The procedure in this section shows how to create a multi-page login form for the Threaded Message Database. You can easily extrapolate this procedure to create custom login forms—and hence additional security—for any sensitive IntraBuilder application.

1. Open the Form Designer. By the way, you can also open the Form Designer directly by typing in the Script Pad:

   ```
   _sys.forms.design()
   ```

2. Drag the USERS.DB table onto the new blank form to create an active Query object.
3. Set the query's rowset's *locateOptions* property to 2 (ignore case).
4. Once again, drag the USERS.DB file onto the new blank form to create a second active Query object and set the its rowset's *locateOptions* property to 2 (ignore case).
5. Set the *active* property of users2 to false.
6. Add the following components to the form:

   1. An HTML component with the *text* "TMD Login" as a title for the form.
   2. An HTML component with the *text* "Registered users login here:".
   3. An HTML component with the *text* "Login name" as a field label. Enlarge the HTML component just enough to accommodate the text, but don't let it overlap other components.
   4. From the Field Palette, drag a Text component for the User name field and place it to the right of the "Login name" label you just added. Because you are dragging this component from the Field Palette (whose components are automatically linked to the table fields by the Query object), it is already *dataLink*ed to the User name field of the active Query object, *users1* and it automatically gets the *name* userName. (Recall that we deactivated the other Query object *users2*.)
   5. An HTML component with the *text* "Password" as a label for the password field.
   6. From the Component Palette, drag a Password component and place it to the right of the corresponding label. Then use the Inspector to set its *dataLink* property to the Password field in users1 and set its *name* to password. (We do it this way, because if you dragged the Password field from the Field Palette you would get a Text component rather than a Password component.)
   7. A Button component with the *text* "Submit login" with the *name* loginButton. Resize the Button control to accommodate the text.
   8. A horizontal Rule object to divide the registered users section from the new users section of the form. If you like, in the Inspector, set the Rule's size property to 4 or 5 to make it a thick divider.
   9. A Button component with the *text* "New users click here" with the *name* newButton. Resize the Button control to accommodate the text.

7. Now we are ready to create page 2 of the TMD Login: the New User Registration, Page One. The purpose of this page is to allow new users to identify themselves to the system. Click the Next Form Page button in the toolbar to display page 2.
8. Add the following components to page 2 of the Login form:

   1. An HTML component with the *text* "New User Registration, Page One" as a title.
   2. An HTML component with the *text* "First name". Enlarge it just enough to display the text so that it won't overlap another component.
   3. From the Field Palette, drag a *dataLink*ed Text component for the First name field and place it adjacent to the corresponding label.
   4. An HTML component with the *text* "Last name".
   5. From the Field Palette, drag a *dataLink*ed Text component from the Field palette for the Last name

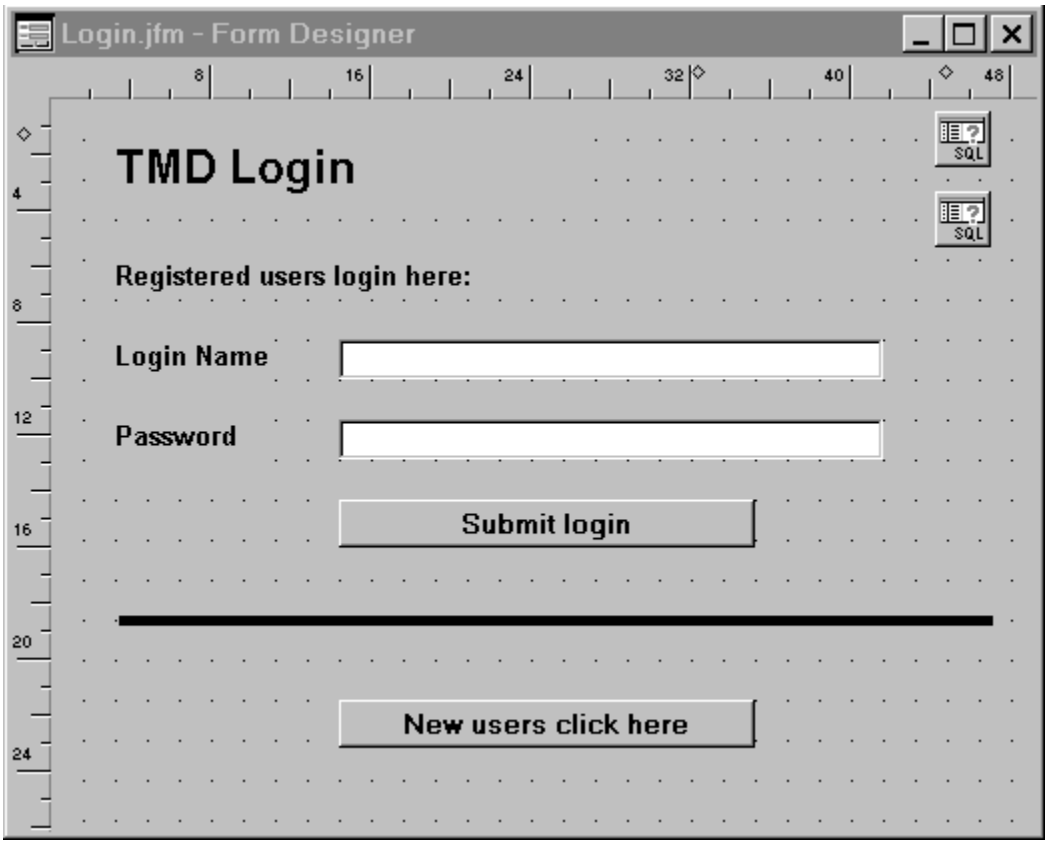
field.

6. An HTML component with the *text* “E-mail address”.
7. From the Field Palette, drag a *dataLink*ed Text component from the Field palette for the E-mail address field.
8. A Button component with the *text* “Register” with the *name* registerButton.
9. A Button component with the *text* “Cancel” with the *name* cancelButton1.

9. Now we are ready to create page 3 of the TMD Login form, the second page of the New User Registration. The purpose of this page is to allow new users to set their user name and password. Click the Next Form Page button in the toolbar to display page 3.
10. Add the following components to page 3 of the TMD Login form:
    1. An HTML object with the *text* “New User Registration, Page Two” for the title.
    2. An HTML object with the *text* “Login name” for a label.
    3. From the Field Palette, drag a *dataLink*ed Text component for the User name field. This is automatically named userName1 because userName is already used.
    4. An HTML object with the *text* “Password”.
    5. A Password component which is *dataLink*ed to the Password field in users1 and has the *name* Password1.
    6. An HTML object with the *text* “Re-type password”.
    7. A Password component which is not *dataLink*ed and has the *name* Password2.
    8. A Button with the *text* “Try login name” with the *name* tryNameButton.
    9. A Button with the *text* “Cancel” with the *name* cancelButton2.
11. Set the form’s *title* property to “TMD Login”.
12. Switch back to page 1.
13. Save the TMD Login form and give it the name LOGIN.JFM, but don’t close the Form Designer.

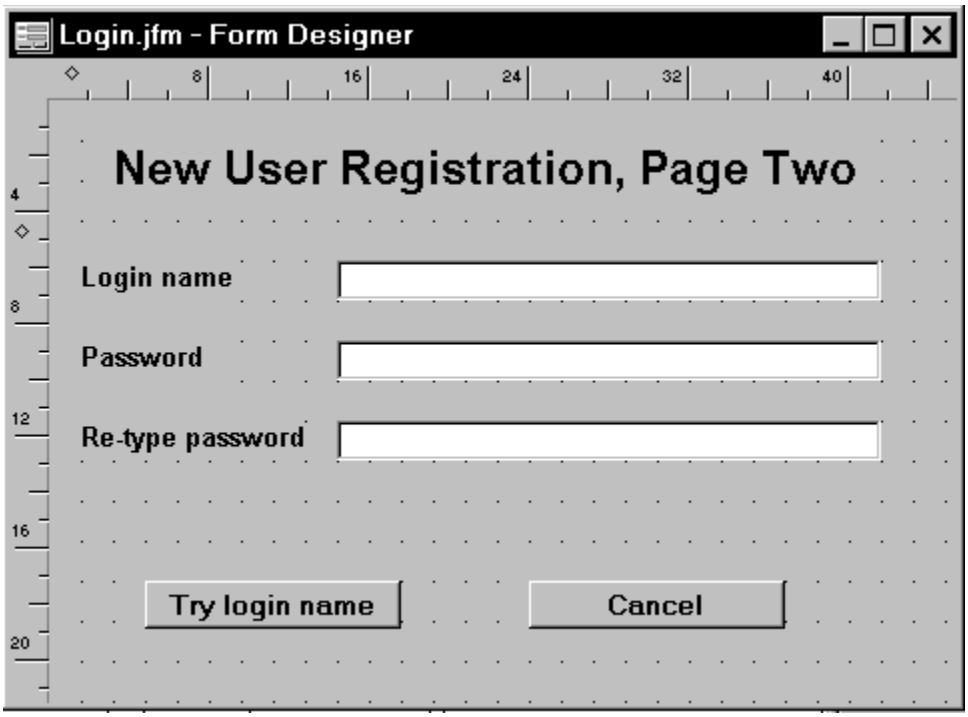
The TMD project: Creating the login form [figures]

Page 1 of the Login form



Page 2 of the Login form

Page 3 of the Login form


Login.jfm - Form Designer
New User Registration, Page Two
Login name
Password
Re-type password
Try login name
Cancel

The TMD project: Registering a new user

Related topics

Now that the components of the form have been laid out, everything has to be hooked together.

The first task is to register new users. New users should click the “New users click here” button. This will take them to the second page, where they can type in the required user information.

Type in the following JavaScript for the newButton button’s *onServerClick* event handler:

```
function newButton_onServerClick()
{
  this.form.users2.active = true;  // Activate 2nd query for lookup
  this.form.rowset.beginAppend();  // Add new user
  this.form.pageno = 2;            // Switch to first new user page
}
```

This code activates the second query which is used for user name lookup. Then it puts the first query’s rowset in Append mode and switches the form to page 2, which is the first page of the new user registration.

On page 2, the new user types in their first name, last name, and E-mail address. You could also get more information by adding fields to the Users table and creating *dataLink*ed components for data entry.

There’s also a cancel button on page 2. If users click it, it cancels the append and switches the form back to page 1. Set its *onServerClick* event handler to:

```
function cancelButton1_onServerClick()
{
  this.form.rowset.abandon();  // Abandon the append
  this.form.pageno = 1;        // Switch back to the login page
}
```

If users click the registerButton, a login name is suggested, which is simply the first name and the last name separated by a space. The application stores this name in the *value* property of the userName1 component on page 3. The non-*dataLink*ed Password component is cleared (in case the user makes multiple attempts to register), and the form is switched to page 3.

In the *onServerClick* event handler of registerButton, enter this method:

```
function registerButton_onServerClick()
{
  this.form.userName1.value = this.form.firstName.value + " " +
this.form.lastName.value;
  this.form.password2.value = "";
  this.form.pageno = 3;
}
```

Moving objects to other pages

Related topics

Page 3 has a Cancel button that does exactly the same thing as the one on page 2. You could have implemented it by moving the button from page 2 to page 3. To do this, set its *pageno* property at the same time you set the form's *pageno* property.

This approach makes more sense if the object has a lot of properties and events you don't want to duplicate.

If you move an object to another page, you'll have to make sure its *top* and *left* properties are updated as well if the object does not appear in the exact same position on each page. Also, if users can go back to the previous page, as they could in this login form, you'll need to move the object back.

But the Cancel button is simple, so it easier to create a separate one for each page.

The TMD project: Linking the same event to other objects

Related topics

Because both Cancel buttons do the same thing, there's no need to create separate event handlers. They can both call the same event handler, in this case from their *onServerClick* events.

This technique can be used not only for objects that do exactly the same thing, but also for objects that do similar things. Because an event is object-oriented, the event handler can use the properties of the object in its actions.

For example, in an address book, you could create buttons for every letter in the alphabet, and when the button is clicked, you can show the first name in your table that starts with that letter. Because each button has the letter on it as the *text* property, the event handler can use that letter in its search.

There are two ways you can link events to objects: in the dialog boxes of the Form Designer or by entering JavaScript in the Script Editor. Both are described in the next sections. For the TMD project, use one of these methods.

Linking events in the Form Designer

Related topics

When you click the tool button on the Events page of the Inspector, it creates a method in the Method Editor and automatically links that method to the event. Or you can create the method first in the Method Editor by using the New Method menu option, and then link the method to the event with the Link Method menu option. (That's also how you link an existing method to multiple objects.)

To link methods to events,

1 Select the cancelButton1_onServerClick method in the Method Editor's drop-down list.

2 Right-click the method to display the shortcut menu. Choose the Link Event option.

3 In the Link Event dialog box, select cancelButton2 from the object list.

4 In the Link Event dialog box, select *onServerClick* from the Event list, and click OK.

The method is now linked to both events in both objects. To the left of the Method Editor's drop-down list are the events to which the method is linked. If there are more events than displayed, a question mark appears when the mouse pointer moves over that area.

Click the question mark over the area to display a pop-up list of the events.

Linking events in scripts

Related topics

If you find yourself in the Script Editor editing a JFM file as a script, it's also easy to assign methods to events. You connect a method to an event by assigning a reference to that method to the object's event property. (If you are editing the form in the Form Designer, close the form, then reopen it in the Script Editor.)

For example, to hook up the cancelButton1_onServerClick event to the second Cancel button in the script instead of the Form Designer, you would first find the *onServerClick* property assignment in the *with* statement that creates the first Cancel button:

```
with (this.cancelButton1 = new Button(this)){
  left = 42;
  top = 7;
  width = 10.5;
  text = "Cancel";
  pageno = 2;
  onServerClick = class::cancelButton1_onServerClick;
}
```

Copy that *onServerClick* line to the *with* statement that creates the second Cancel button:

```
with (this.cancelButton2 = new Button(this)){
  left = 40;
  top = 8;
  width = 10.5;
  text = "Cancel";
  pageno = 3;
  onServerClick = class::cancelButton1_onServerClick;
}
```

When manually assigning properties in a script, it generally doesn't matter where you assign individual properties in the *with* block. However, it isn't a bad idea to follow the same order as the Form Designer. One exception is the Query object's *active* property, which should be assigned last so that the properties that define its rowset can be set before the query is activated.

Renaming methods

Related topics

Because the same method is assigned to both Cancel buttons, its name, cancelButton1_onServerClick, doesn't quite fit. This might cause confusion as to whether it should be called by cancelButton2.

To change the name of an event, use the form designer to display the method in the Method Editor and change its name as it appears in the editing area. Any object's events linked to that method will automatically be updated if you choose another event or close the form. If you call that method from other method, the names are not changed, because this is code that you typed.

You can also change the method name when editing the form as a script by using the Script Editor's search and replace function. If the method name is used as part of other names, selecting the Whole words option will help differentiate between them.

Now, we want to rename a method in the Login form:

Change the method name from cancelButton1_onServerClick to cancelNew.

Using search expressions

Related topics

On page 3, users can accept the suggested name or type in their own. They also need to type in their desired password twice to make sure they don't make a typo. When they click the "Try login name" button, the first thing to check is whether the user name is already in use. (In fact, the login form should check to see if the name is in use before suggesting it. If it is, change the name by adding a number to it.)

Because the first query is in Append mode to store the data for the new user, the login form must search the second query, which was activated when the user clicked the "New Users Click Here" button. Theoretically, you could do a *beginLocate*() in three statements like this:

```
this.form.users2.beginLocate();
this.form.users2.rowset.fields[ "User name" ].value =
this.form.userName1.value;
if ( this.form.users2.applyLocate() ) {
  // The user name is already in use
}
```

Another approach is to use a SQL expression in the call to the *applyLocate*() method.

SQL expressions

Related topics

SQL expressions use field names, literal data values, operators, and a few other options like AND and UPPER() to describe a condition. These options are listed in the *Language Reference*.

For example, to look for a particular user name, the SQL expression would be

```
"User name" = 'Joe Bob'
```

Note that the field name is enclosed in double quotation marks and the literal string is enclosed in single quotation marks. This is the preferred SQL syntax. If the field name did not have a space or some other special character in it, it wouldn't have to be enclosed in quotation marks. Also, in SQL the comparison operator is a single equals sign, not the double equals used in JavaScript.

Because the SQL expression itself is passed as a string, you must juggle all the quotation marks, as shown in the next section.

Building the SQL expression

Related topics

The call to *applyLocate*() looks like this:

```
this.form.users2.rowset.applyLocate( '"User name" = \'' +
     this.form.userName1.value + '\'' )
```

Note Because JavaScript doesn't care about line breaks, you can break a long line into separate statements, as shown here.

Because the User Name field is enclosed in double quotation marks, the string uses single quotation marks as the string delimiters. To put a single quotation mark in the string, it must be preceded by the backslash (the escape character). Next, add the *value* of the userName1 component, which contains the new user name, followed by the ending single quotation mark.

(This single quotation mark could have been placed inside double quotation marks to avoid using the backslash, but then using both single and double quotation marks to delimit strings in the same statement is probably even more confusing.)

Instead of using the *value* of the component on the form, you could have used the *value* of the field in the rowset, because the component value has been echoed into the field through the *dataLink* property. The main drawback is that referencing a field from a button's event handler requires more typing:

```
this.form.rowset.fields[ "User name" ].value
```

If you're building a SQL expression with a numeric literal, numeric literals do not have quotation marks; for example:

```
this.form.rowset.applyLocate( '"Message #" = ' +
     parseInt( this.form.rowset.fields[ "Reply to" ].value ) );
```

This is actually used later. The *parseInt*() function is used to make sure the number appears as an integer in the SQL expression.

Using non-dataLinked components

Related topics

If the user name has not been used, then the next step is to verify that the password was typed in the same way in both Password components on page 3. The first Password component is *dataLink*ed to the Password field so that it will be saved when the row is saved. The second Password component is not *dataLink*ed to anything. It's simply a Text component the user can type into and it will maintain its *value*.

Because a non-*dataLink*ed component doesn't have a corresponding field, your code must refer to its *value* via the component. Although *dataLink*s are certainly more convenient, there's nothing to prevent you from doing data entry into non-*dataLink*ed components and then manually copying the value into field objects. You might do that if you were creating multiple rows with the same data for example. The opposite is also possible: reading data from the rowset and manually updating the *value* properties of the components on a form. Given the various events and other data access capabilities in the data classes, however, these cases are unlikely.

A more likely use of non-*dataLink*ed controls is when the input is not destined to be stored in a table. For example, the act of logging in takes a user name and password and simply looks them up in table. You could use non-*dataLink*ed components and use *applyLocate*().

Just to show an interesting way to do a login, *dataLink*ed components are used on page 1, the login page. However the new user verification must be completed first.

Using exceptions to manage script execution

Related topics

The new user verification involves two steps:

1 Make sure the user name is not already used, and if not, then

2 Make sure the password was typed in the same both times

Theoretically, you could code this by using nested *if* statements:

```
if ( this.form.users2.rowset.applyLocate( '"User name" = \'' +
     this.form.userName1.value + '\'' ) )
{
  // Name already in use
}
else {
  if ( this.form.password1.value == this.form.password2.value ) {
    this.form.rowset.save();  // Save the new user
    this.form.pageno = 1;     // Switch back to login page
  }
  else {
    // Passwords don't match
  }
}
```

While the above nested *if* statements would work, you would have two different error handlers at different nesting levels, and the code to execute upon success is nested at the lowest level. In this case, with only two levels it's not a problem, but if there were more conditions, things would be much messier. Also, if the error handling gets complicated, you would either have a lot of redundant code strewn about or you might have to set up error flags.

A better approach—especially in this situation with a number of tests, of which all must succeed in series—is to use exception handling. Although exception handling is used to handle errors in critical sections of code, you can also generate exceptions yourself to manage script execution. Each test is run one at a time at the same level of code. If the test fails, the code throws an exception with the *throw* statement, which is handled by a single *catch* exception handler. If all the tests pass, then the final code executes.

The TMD project: New user verification

Related topics

Now we're going to implement user verification in the TMD project; we'll check that the user name has not been used by someone else and that the new user has typed the password correctly.

Here is the new user verification code using exception handling. Type this code as the tryNameButton buttons's *onServerClick* event handler:

```
function tryNameButton_onServerClick()
{
  try {
    var e = new Exception();  // Create Exception object in case you need to
throw
    if ( this.form.users2.rowset.applyLocate( '"User name" = \'' +
        this.form.userName1.value + '\'' ) )
    { // If user name is already used
      e.message = "Login name already in use";
      throw e;  // set the error message and throw the exception
    }
    if ( this.form.password1.value != this.form.password2.value ) {
      e.message = "Passwords do not match";
      throw e;  // Same if passwords don't match
    }
    // If you get this far, there were no errors
    this.form.rowset.save();  // so save the new user
    this.form.pageno = 1;     // and switch back to the login page
  }
  catch ( Exception e ) {     // When there's an exception
    this.text = e.message;    // Set the button's text to the appropriate
message
  }
}
```

In contrast to the nested *if* statements, this code is linear, and can easily take more conditions.

The statements that might throw the exception are placed in a *try* block. If the code executes correctly then no exception is generated.

First, if you're going to throw an exception, you will need to create an Exception object. An Exception object has a numeric *code* property and a *message* string property. Both these properties are set when IntraBuilder generates its own exceptions, usually in response to an error.

If a test fails, you can set the Exception object's *message* property, or any other property for that matter, and then *throw* the Exception object. Execution jumps to the *catch* block and takes that object as a parameter. In the code example, both the Exception object that is thrown and the parameter are named e, but that's just a shorthand convention. The two don't have to be the same. You could also create multiple Exception objects with different property settings and *throw* the one you want.

Once an exception is thrown, any code after the *throw* in the *try* block is skipped. If there is a corresponding *catch* handler, the exception is handled and discarded. If not, then the exception bubbles up through any nested *try* blocks. If the exception is not caught by a *catch*, eventually it gets to IntraBuilder itself, which generates an error dialog box, using the Exception object's *message* property as the error message.

In addition to *try* and *catch*, there is also a *finally* option in exception handling. These are explained in more detail in the *Language Reference*.

At this point, save and run the login form. You should be able to add new users. Make sure that the new user verification is working by trying to enter the same user name twice and trying different passwords. When a new user is successfully added, the form goes back to the main login screen—which doesn't work yet.

Site access passwords

Related topics

When users type in their user names and passwords, these strings are checked against a table of names and passwords. If either user name or password doesn't match—the name is not listed, or the password does not match the name—the login attempt fails. If the login fails, you should report to the user only that it failed, and not why. Limiting feedback to the user attempting access will hamper attempts to break into your system.

This type of login is managed entirely by your code. It is intended to provide secure access to your application; either certain parts or its entirety.

IntraBuilder provides automatic table login forms. These are explained in the Security section of the Help file, and should not be confused with the type of login form you're creating here.

The TMD project: Password lookup

Related topics

We will now look up the submitted user name and password in the USERS.DB table.

To lookup the name and password, you could use non-*dataLink*ed controls and insert their *value*s into a SQL expression in an *applyLocate*(), call:

```
if ( this.form.rowset.applyLocate( '"User name" = \'' +
this.form.userName.value +
     '\' and Password = \'' + this.form.Password.value + '\'' ) )
```

However, you would have to type that in and get everything right.

Here's an easier way:

1. Use components *dataLink*ed to the User name and Password fields.
2. Put the rowset in Locate mode and let the user type in their user name and password.
3. Use this *if* statement:

   ```
   if ( this.form.rowset.applyLocate() )
   ```

Besides just being shorter, this second approach is more flexible. If you change, add, or remove fields, you have only to deal with the component and the *dataLink*, both of which you can do visually. You don't have to modify code that uses both the field name and the reference to the component, along with any ANDs or lack thereof in the SQL expression string.

The trick to making this approach work is to mark both the User Name and Password fields as modified by "touching" them, that is, by assigning an inconsequential value just to force marking the field as modified. Because both fields will be considered as modified, they will both be used in the search. If both fields were not touched, someone could type just the user name and get a match on that alone, without having to type in a password.

To touch the User Name and Password fields, you assign inconsequential values to both fields after the call to *beginLocate*().

1. Place the code to do this in a method, because it is called from a number of places. Open the Login form in the Form Designer
2. Go to the Method Editor. Right-click in the editing area to display the shortcut menu. Choose New Method. This creates the following method skeleton:

   ```
   function Method()
   {
   // {Export} This comment causes this function body to be sent to the client
   }
   ```

   The // {Export} line is used to export client-side JavaScript methods. This is explained in Exporting JavaScript methods . Because this method is run on the server-side only, delete this line.
3. Fill in or replace the method skeleton with this method:

   ```
   function beginLogin()
   {
     this.rowset.beginLocate();                          // Switch to Locate
mode for login
     this.userName.value = this.password.value = "" ;    // Touch fields to
require entry
   }
   ```

   Notice that you can use a single statement to assign the same value to multiple locations.
4. Call this method when the form opens, which is explained in the next section.

Running code when opening a form

Related topics

Forms and components have an *onServerLoad* event that fires when they are loaded, that is, opened, on the server. The form's *onServerLoad* event is a good place to call the beginLogin() method.

All methods created in the Method Editor are methods of the form. In the form's *onServerLoad* event, the reference *this* refers to the form, so you call the beginLogin() method through *this*.

The TMD project: Activating the login form

Related topics

Now we're going to call the *beginLogin*() method. In the TMD Login form's *onServerLoad* event, enter this method:

```
function Form_onServerLoad()
{
  this.beginLogin();
}
```

Running another form

Related topics

To run another form, call the *run*() method of the *_sys.forms* object.

In the run() method, enter a parameter value for the form name, along with the form's parameters. For the form name, a JFM extension is assumed, along with any optional parameters. In this case, the VIEWER.JFM is run, and a reference to the form is passed as a parameter.

The TMD project: Running the Viewer form

Related topics

The last step to actually make the TMD login form work is to try the user name and password and run the Viewer form if the user name and password are found:

Type in the following *onServerClick* event handler for the loginButton:

```
function loginButton_onServerClick()
{
  if ( this.form.rowset.applyLocate() ) {
    _sys.forms.run( "VIEWER", this.form );
    this.form.close();
  }
  else {
    this.text = "Try again";
    this.form.beginLogin();
  }
}
```

If a match is not found, the text of the button is changed, and the beginLogin() method is called again so that the user can try another user name and password.

The TMD project: Finalizing the login form

Related topics

Because the TMD login process uses Locate mode, there are a few changes that must be made to the existing code.

First, if the new user button is clicked, abandon the Locate mode before switching into Append mode for the new user. Add the highlighted lines to the existing code:

```
function newButton_onServerClick()
{
  this.form.rowset.abandon();       // Cancel Locate mode for login
  this.form.users2.active = true;   // Activate 2nd query for lookup
  this.form.rowset.beginAppend();   // Add new user
  this.form.pageno = 2;             // Switch to first new user page
}
```

If the user cancels the New User operation, switch back into Locate mode and touch the fields:

```
function cancelNew()
{
  this.form.rowset.abandon();   // Abandon the append
  this.form.beginLogin();       // Go back to Locate mode and touch the fields
  this.form.pageno = 1;         // Switch back to the login page
}
```

Finally, after entering a new user name and login, there's no need to return newly registered users to page 1 of the TMD Login. Instead, take them directly to the Viewer form. To do this, add the two highlighted lines and remove the strikethrough line:

```
function tryNameButton_onServerClick()
{
  var e = new Exception();  // Create Exception object in case you need to
throw
  try {
    if ( this.form.users2.rowset.applyLocate( '"User name" = \'' +
         this.form.userName1.value + '\'' ) )
    { // If user name is already used
      e.message = "Login name already in use";
      throw e;  // set the error message and throw the exception
    }
    if ( this.form.password1.value != this.form.password2.value ) {
      e.message = "Passwords do not match";
      throw e;  // Same if passwords don't match
    }
    // If you get this far, there were no errors
    this.form.rowset.save();  // so save the new user
    _sys.forms.run( "VIEWER", this.form );  // and login
    this.form.close();
    ~~this.form.pageno = 1;      // and switch back to the login page~~ remove
this line
  }
  catch ( Exception e ) {     // When there's an exception
    this.text = e.message;    // Set the button's text to the appropriate
message
  }
}
```

Form cosmetics

Related topics

When you run the login form locally, you may notice that the first user name and password are actually displayed momentarily on page 1 in the *dataLink*ed components. This is because the *onServerLoad* event that puts the rowset in Locate mode fires after the form has been opened.

This is not a problem when logging in through a browser, because the form will be not sent until all the events are done.

But if you want to fix the problem locally, one easy solution is to save the form with its *pageno* property set to a blank page, like 100, so that the form will open there. Then in the *onServerLoad*, call the beginLogin() and then switch to page 1.

A more persistent problem is the fact that the Form Designer displays the contents of the first row when editing a form. So you might want to put a dummy or test user name and password as the first row in the table.

The forms stack

Related topics

When forms run other forms, each form is managed internally by IntraBuilder in a form stack. The classic analogy for stacks is a stack of dishes in the cafeteria. Running a form is like placing a new dish on the top of the stack. The form on the top of the stack is the one that is displayed. If that form is closed, it's like removing the dish from the top of the stack; the one underneath is then the one that is shown.

Unlike most stacks in programming, you can remove a form from the middle of the stack by closing it. If all the forms in the stack are closed, IntraBuilder will send an error message to the browser.

In the TMD project, the user starts in the login form. This is the only form on the stack. If the user successfully logs in, then the Viewer form is run. It goes on the top of the stack and is displayed on the browser.

Next, the login form is closed. This removes the login form from the stack, but the user doesn't know this, because IntraBuilder sends only the form on the top of the stack.

If the login form were not removed and there were a way to close the viewer form, the user would go back to the login form. But there is no HTML button nor a window close box.

Although the user can go back in the browser to the login form, the form won't work, because the browser is simply displaying a cached copy of the form. The form on the IntraBuilder server is already closed, so any communication with it has been lost.

Displaying unlinked data

Related topics

The easy way to display data from a table is with a *dataLink*ed component. This automatically updates the data as you navigate and allows editing of the data.

Sometimes you don't want to allow editing. In the TMD, you don't want users to edit the Reply to, # of replies, and Posted date fields. These are informational only.

So instead of using *dataLink*ed components, use HTML components and set their *text* properties in the *onNavigate* event of the rowset.

The TMD project: Displaying message information

Related topics

Follow these steps:

1. Open VIEWER.JFM in the Form Designer.
2. Delete the Text component that displays the Reply to field.
3. Change the *name* of the Reply to label from HTML6 to replyToLabel and set its *pageno* property to 1.
4. Move it next to the right of the Subject Text component.
5. The intent of the TextArea component for the Body of the message is fairly obvious, and space to the left of it would be put to better use making the TextArea as big as possible, instead of for the field label. Delete the HTML component with the word “Body” and resize the TextArea so that its left edge is flush with the other field labels.
6. Create an HTML component under the Body field’s TextArea component. Set its name to numRepliesLabel and its *fontBold* property to *false*.
7. Create an HTML component to the right of the message #. Set its name to dateLabel and its *fontBold* property to *false*.
8. Create a new method named refreshUnlinked(). This method updates the *text* property of the HTML components as the user navigates through the messages. It displays non-editable information.

   Type in this code:

```
function refreshUnlinked()
{
  if ( this.rowset.endOfSet ) {
    this.replyToLabel.visible = false;
    this.numRepliesLabel.text = "";
    this.dateLabel.text       = "";
  }
  else {
    if ( this.rowset.fields[ "Reply to" ].value > 0 ) {
      this.replyToLabel.text    = "Reply to " +
                                  parseInt( this.rowset.fields[ "Reply
to" ].value );
      this.replyToLabel.visible = true;
    }
    else {
      // Thread root message, not a reply
      this.replyToLabel.visible = false;
    }
    this.numRepliesLabel.text = "Replies: " +
                                parseInt( this.rowset.fields[ "# of
replies" ].value );
    this.dateLabel.text       = this.rowset.fields[ "Posted" ].value;
  }
}
```

   It’s important to check the rowset’s *endOfSet* property, because attempting to access a field at the end-of-set would cause an error. If the rowset is at the end-of-set, the unlinked components are blanked.

9. Set the rowset’s *onNavigate* event to:

```
function messages1_onNavigate()
{
  this.parent.parent.refreshUnlinked();
}
```

10 Before we forget, set the title property of the form to “TMD Viewer”.

By the way, the HTML objects will be updated when you run the form, so the user will never see the “HTML6”. But don’t run the form now, because it won’t work yet.

The TMD project: Incorporating the user ID in the Viewer form

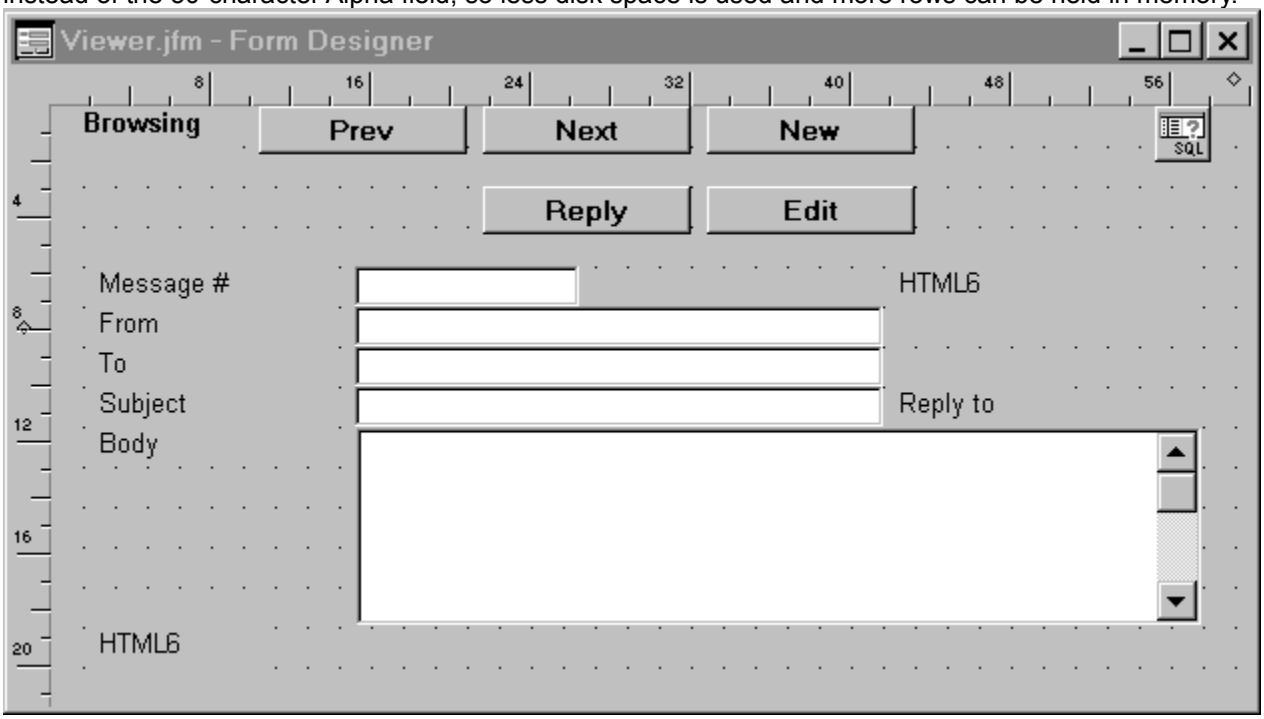
Related topics

As originally designed, the Viewer form simply allowed direct entry of a name into the From field. The Viewer did not know who the user was, so it could not automatically fill in the field. Moreover, one user could easily impersonate another.

By adding the login form, each user is now identified, which solves both of those problems. But because each user is now identified by a unique ID number, the TMD can use that number instead of the name.

This has a couple of immediate advantages:

- If users change their user name (get married, religious conversion, witness protection, and so on) they won't lose ownership of their messages and the marking of those they've read. There is currently no option to change their name in the login form, but it would be easy to add.
- The user ID number is considerably shorter than the name, 4 bytes for the AutoIncrement field instead of the 30-character Alpha field, so less disk space is used and more rows can be held in memory.



Viewer form with unlinked text components.

Key fields in tables

Related topics

A more subtle but in some ways more important point is that when linking two tables together in a master-detail relationship—like customers-orders, teachers-classes, or messages-read (which will be implemented later). The link should use key fields that you, the developer, control.

For example, suppose you used the 7-digit home phone number as a unique identifying key field for your local customers. This might seem good enough, but what if the person changes their phone number? You would lose all the links to data you have on file. A bigger problem would be if the phone company ran out of phone numbers and had to add a new area code. Then you would have to change all your tables, forms, and code from 7-digit numbers to 10-digit numbers.

If you had used a key field which you generate and control, you wouldn't have either of these problems. Just make sure your key field is big enough to accommodate all your rows. For example, a four-byte AutoIncrement field can handle over four billion rows. If the table format you're using does not support byte-level fields you can estimate how many you'll have and then add an extra character or digit to be on the safe side. For example, if you have thirty employees, two digits would be enough, but you might as well make it three or four in case the company grows.

Even though you shouldn't use a field like phone number for a key field, you can still use it as a lookup field. Customers probably won't know their ID number, and in many implementations, like the TMD project, the ID number is never displayed, although it is pervasive throughout the underlying code and tables.

Receiving parameters to forms

Related topics

When the Login form runs the Viewer form, it sends the User ID as a parameter. The Viewer form takes that parameter and stores it so that it knows who the user is. When a form is run, it can receive any parameters in a number of ways, because it is a script file with a .JFM extension.

arguments array

Related topics

Whenever a function, method, or script file is called, an object is created that contains an array. This array lists any parameters or arguments passed in the call.

The object has the same name as the function. For the main code in a script file that is not in a function or class, the name of the script file is used, without the extension and in all uppercase. For example, when VIEWER.JFM is run, an object named VIEWER is created. Remember that JavaScript is case-sensitive.

The function object has a single property, *arguments*, which refers to an array. This *arguments* array has a *length* property, which indicates the number of arguments that were passed to the function or script. If *length* is greater than zero, then each argument is stored as an element in the array, starting from element number zero. For example, if two arguments are passed to a function, then *length* is 2 and the arguments are stored in *arguments*[0] and *arguments*[1].

Secure parameter passing

Related topics

The form's *onServerLoad* event is the most convenient place to receive parameters when using the Form Designer. Set the event handler to the following (but don't try and run it, because there's no userLabel object yet):

```
function Form_onServerLoad()
{
  try {
    // Get user ID and name from login form
    this.userID   = VIEWER.arguments[ 0 ].rowset.fields[ "User ID"   ].value;
    this.userName = VIEWER.arguments[ 0 ].rowset.fields[ "User Name" ].value;
  }
  catch ( Exception e ) {
    this.userID   = 0 ;              // Eventually will not allow access
    this.userName = "Unregistered" ; // But for now allow unregistered user
  }
  // Set "From" name on compose page to user name, because it will never
change
  this.userLabel.text = this.userName;
  this.refreshUnlinked();
}
```

This method uses exceptions to handle the secure parameter passing between the calling form and receiving form. A number of things could fail, and if they do, it's probably because the form was run directly from the Explorer or the browser. For testing purposes, you can allow this and set the User ID to zero and the name to "Unregistered", but eventually you'll want to prevent access to the TMD.

The first statement in the *try* block can fail in three places. As far as the exception handler is concerned, it doesn't matter why the login failed; it just does. The first part of the property reference refers to the first parameter passed to the VIEWER.JFM script:

```
VIEWER.arguments[ 0 ].rowset.fields[ "User ID" ].value
```

If there were no parameters, then that would be an exception.

Next, the parameter is assumed to be an object reference, specifically to one with a *rowset* property:

```
VIEWER.arguments[ 0 ].rowset.fields[ "User ID" ].value
```

Form and Query objects have a stock *rowset* parameter, but it's certainly possible for other objects to have one. If the parameter is not a reference to an object with a *rowset* property, then that's an exception.

Finally, the rowset must have a field named "User ID" in its *fields* array, and that field has to have a *value* property:

```
VIEWER.arguments[ 0 ].rowset.fields[ "User ID" ].value
```

Again, it's an exception if something doesn't match up.

The whole point of this approach is to get the verified user ID from the login form. The login form could have obtained the ID from its rowset and passed that as a parameter instead of the form reference. It does not, however, because the viewer form would be expecting a number as a parameter. The danger there is that unauthorized users can include any number as a parameter when they call a form in their browser.

So while you would want users to call the LOGIN.JFM form and let it call the VIEWER.JFM form with the user ID, they could call VIEWER.JFM directly and include an ID number or any other simple parameter in the URL.

Using a form reference makes the parameter passing secure. An unauthorized user cannot forge a form reference as easily as they could an ID number because there is no literal representation of an object reference.

Denying access to a form

Related topics

There are a number of ways of denying access to a form after it has been loaded. Here are a few:

- Close the form.

If you call the form's *close* method() during its *onServerLoad* event, the user will never see it, because it is closed before it has a chance to be transmitted to the browser. It would be as if the form never opened.

- Switch to another page on the form.

Create a page that contains the message "Access denied" Switch to that page in the *onServerLoad* event. As long as you don't include any buttons that would allow the user to switch to another page, that's all they will see. However, if the form has components on page 0, those appear on all pages. Therefore you would have to set all the *visible* properties of those components to false; otherwise they would appear on the "Access denied" page.

- Open another form.

This form contains an "Access denied" message. Unlike traditional multi-window data entry applications or Windows itself, users cannot switch between forms from their browser. They see only the form that is on the top of the form stack, which you, as the developer, control.

Form variables

Related topics

Because an *applyLocate*() is used for the match in the login form, its rowset would be positioned at the user's row, which contains their ID. The Viewer form extracts the ID number from the form object and assigns it to a property of the form.

The userID property is not a stock property; it is created in the *applyLocate*() statement. JavaScript's dynamic object model allows you to add properties whenever you want. The main drawback is that you might inadvertently add a new property when you wanted to assign to an existing one—due to misspelling or mis-capitalizing the property name.

By storing a variable as a property of the form, it persists as long as the form does, which is usually all you care about, because the form represents the link between the browser and IntraBuilder.

In contrast, a plain variable lasts only until the function that created it terminates. In this case, that would be when the *onServerLoad* is done. That would mean that the user ID would not be available to any other events that occur after the form has completed loading.

The user's name is also retrieved from the login form and stored as the form variable userName.

The TMD project: Displaying the From name

Related topics

Now we want to program the TMD to automatically enter the user's name in the From field of a new message.

After the *try*/*catch* is used to set the userID and userName form variables, the *text* property of the form's userLabel component is set to the userName. Therefore we use the userLabel as the From name on the Editing page (page 2). Because any message composed by the user will use their name, there's no need to use a Text component in which users must type their name; an HTML text object will suffice.

If the userLabel is on the Editing page, then the Text component for the From name must be moved from page 0 to page 1, the Browse page:

1. In the Form Designer, select the From Text object that was created by the Form Expert.
2. In the Inspector, change its *pageno* property from 0 to 1.

To create the userLabel,

1. Switch the form to page 2
2. Drag an HTML component from the Component palette and place it above the To Text object, making sure their left edges are aligned. The easiest way to do this is to make sure the userLabel is a bit to the right of the To object, select both of them, and click on the Left align icon in the speedbar.
3. Set its *name* property to userLabel.
4. Switch the form back to page 1.

By the way, the form's refreshUnlinked() method is called at the end of the *onServerLoad* event handler to update the informational text that was created in Displaying unlinked data when the form first opens.

One-way field morphing

Related topics

The From field in the Messages table now stores the user ID number instead of the user name. This is good because the ID number can be used to streamline the application, resulting in better performance. However, at the moment the From Text object is *dataLink*ed to the From field, so the From field of the Editing page displays that user ID number—which isn't very user friendly.

Instead of a *dataLink*ed Text component, you could use an HTML object and do a lookup on the user ID number in the *onNavigate* event to update it with the user name. This would be a slow, repetitive way.

But there's a better way: A Field object's *beforeGetValue* event is fired whenever a field's *value* property is referenced by your code or through a *dataLink*. The event handler can *return* any value, and that value is used as the field's *value* property. This is referred to as "field morphing." Thus, the actual field contains one value (in this case a user ID number) but its *dataLink*ed component displays another mapped value (in this case, the user name associated with the user ID).

To create a one-way field morphing,

1. In the Form Designer, first add the USERS.DB table to the form by dragging it from the Explorer onto the form. This creates and activates a query named users1.
2. In the Messages query, set the From field's *beforeGetValue* event. To get to a field in the Inspector, use the object drop-down list to select the query's *fields* object. Then expand the Array Elements heading to list the fields. Choose the desired field and click the tool button.

   Type in the following event handler, which handles three separate cases:

```
function messages1_from_beforeGetValue()
{
  if ( this.parent.parent.endOfSet ) {
    // When navigating to end-of-set
    return null;
  }
  else if ( this.value == null ) {
    // For beginAppend()
    return "";
  }
  else {
    // Normal lookup, with value in case lookup fails
    var r = this.parent.parent.parent.parent.users1.rowset;
    return r.applyLocate( '"User ID" = ' + parseInt( this.value ) ) ?
           r.fields[ "User name" ].value : "Unregistered";
  }
}
```

Basically, this code tells the event handler *beforeGetValue*: "Before you return a value from this field, do something else first and return that result instead."

The event handler *beforeGetValue* fires whenever the field is accessed, even when navigating to the end-of-set. You should always test the rowset's *endOfSet* property in a *beforeGetValue* event handler. If you're at the end-of-set, you can return *null*. This is important for two reasons: *beforeGetValue* usually uses the actual *value* of the field. Also, attempting to access a field's *value* at the end-of-set causes an error.

The *beforeGetValue* event also fires when you start a new row with *beginAppend*(). In a *beforeGetValue* event handler, *this.value* contains the actual contents of the field in the table. For a .DB table, the *value* for a new field is *null*, so the *beforeGetValue* event handler returns an empty string.

Finally, for a normal lookup when accessing the field, *applyLocate*() is called for users1, the query that accesses the Users table. It's a bit of a trip from the From field's *beforeGetValue* event to *applyLocate*():

- In the field's event handler, *this* refers to the field
- The field's *parent* is the *fields* array.

- The *fields*' *parent* is the rowset.
- The rowset's *parent* is the query.
- The query's *parent* is the form.
- Through the form, you can access all the queries, like users1.
- Like all queries, it has a *rowset*.
- The *applyLocate*() method is a method of the rowset.

The SQL expression passed to *applyLocate*() searches for a match on the User ID field. The conditional operator (?:) will then return a value depending on whether the search was successful. For a registered user, the search should always be successful, and the User name will be returned. If the search is unsuccessful, "Unregistered" is returned.

Because the From field is a morphed field, the Text component that displays its contents (on page 1) should have its *template* property set to the represent the maximum length of the field. In this case, the User name that is used in the lookup is 30 characters long, so set the *template* property to a string with 30 "X"s:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

In the TMD project, this one-way field morphing is useful in the following situations:

- The *dataLink*ed component that displays the From field on the Browse page of the form, and
- When the From name is copied into the To field for replies. The code in replyButton_onServerClick() which accesses the From field's *value* doesn't have to change, because the *beforeGetValue* event makes the *value* appear as something else.

Field morphing in the other direction, when there's an attempt to store a value in a field, isn't required for the From field. The actual user ID number will be stored directly, because that number is known and doesn't change; you don't have to use the name to look it up. Now that the user's identity is known, you can automatically set the From field when composing replies and new messages.

Add the highlighted lines to the messsages1 query's rowset's *onAppend* event handler:

```
function messages1_onAppend()
{
  this.fields[ "From"         ].value = this.parent.parent.userID;
  this.fields[ "# of replies" ].value = 0;
  this.modified = false;
  this.parent.parent.pageno = 2;  // Display the editing page
}
```

This code uses the userID form variable, which is two *parent*s removed from the rowset, as is the form's *pageno* property. It also initializes the # of replies field to zero (by default it contains a null).

The end result is that now the From field will display the preferred value, the user's name, rather than the actual stored value, the user's ID number.

Replies and threading

Related topics

When users reply to a message or add a new one, and that message is saved, they are currently left at that new message. The TMD interface would be more elegant if it remembered the last message the user was looking at (or replying to) and displayed that message after the user added the new one. To implement this behavior we will use bookmarks.

Bookmarks

Related topics

A bookmark is a server-side data type that represents a row position in a rowset. A rowset's *bookmark*() method returns its current position. To return the user to that row, call its *goto*() method with the bookmark as a parameter.

A bookmark is guaranteed to be valid only as long as the rowset stays open. If the rowset is closed, a saved bookmark might not work; it might take the user to another row or generate an error.

Because the bookmark is needed only when adding replies or new messages, the rowset's *canAppend* event is a good time to save it, because that fires while still on the old row, as opposed to *onAppend*, which fires after the new row has been started. Set the event handler for the messages1 rowset to:

```
function messages1_canAppend()
{
  this.parent.parent.bookmark = this.bookmark();
  return true;
}
```

This gets the bookmark from the rowset and saves it as a form variable. You could save a variable like this in the rowset itself—if you used another property name besides bookmark, because that's the name of the rowset's method. Storing all persistent variables as properties of the form makes things easier to track and keeps form objects centrally located.

The *canAppend* event handler then returns true so that the append can proceed.

When the user saves the new message, you want to return to the *bookmark*ed row:

1. Set the Save button's *name* to saveButton.
2. Set the button's *onServerClick* event to the following method (replacing the Expert-generated codeblock):

```
function saveButton_onServerClick()
{
  if ( this.form.rowset.modified ) {
    if ( !this.form.rowset.save() ) {  // If changed row is invalid
      return;                          // do no more to allow fixes or
abandon
    }
    if ( this.form.rowset.fields[ "Reply to" ].value > 0 ) {
      // Reply
      this.form.rowset.goto( this.form.bookmark );        // Goto original
      this.form.rowset.fields[ "# of replies" ].value++;  // Increment
reply count
      this.form.rowset.save();
    }
    else {
      // New message
      this.form.rowset.fields[ "Thread root" ].value =
           this.form.rowset.fields[ "Message #" ].value;  // Set thread
root
      this.form.rowset.goto( this.form.bookmark );        // then goto
original
    }
  }
  else {                                                  // Nothing
happened, so just
    this.form.rowset.goto( this.form.bookmark );          // goto original
  }
  this.form.refreshUnlinked();
```

```
    this.form.pageno = 1;
  }
```

First, the method checks the rowset's *modified* property to see if the row has been touched.

If so, then the *save*() method is called.

- If the *save*() returns false, then that means the *canSave* failed, so you don't want run any more of the code to allow the user to fix the invalid row.
- If the save is successful, the method determines whether the new message is a reply by checking its Reply To field, which is pre-filled with the parent message number when composing a reply:
- If the message is a reply, the method goes back to the original message with the bookmark, its # of replies field is incremented, and it is saved.
- If the message is not a reply, then its Thread root field is set to its Message #.

The Thread root is used sort the threads. All messages on the same thread have the same Thread root. Because the Message # is an AutoIncrement field, its value is not known until the row is saved. After the Thread root field is set, the bookmark is used to go back to the message which was being viewed. Because the *goto*() method causes navigation, the change to the Thread root field is automatically saved.

Using events versus coding around methods

Related topics

If it turns out that the rowset was not modified, then pressing the Save button still takes the user back to the bookmarked row. (The new row is discarded, because it was not modified.) This illustrates the following subtle point.

Some events occur in response to a user interface action, like clicking on a button. Other events occur when a method is called. This is especially true for the Rowset object, which has a number of can- events that determine if the method's action will actually occur, and on- events that fire after the action.

When those kinds of events are tied to methods, you can achieve the same results as the can- and on- events by coding conditions before actually calling the method and coding statements after the method.

The code in the saveButton_onServerClick() does this. Instead of using the *if* block to check the *modified* property, you could have relied on events. The code to handle the reply or new message could have been placed in an *onSave* event handler, and the *save*() method could be called directly. *save*() automatically checks the *modified* property, and if the row were not modified, the *save*() call would have no effect, and the *onSave* would not have fired.

Events have a number of advantages. For one thing, they centralize code. For example, the rowset's *onAppend* event pre-fills the From field with the user's ID number, whether that new row is a reply or a new message. Without the *onAppend* event, you would have to duplicate the pre-fill code in both the reply and new buttons' *onServerClick* events after calling *beginAppend*(). The Form Expert and Designer also lean toward using events. The Expert generates buttons that call rowset methods in a codeblock. To augment those methods, it's simpler to program their events than to change the button.

So why does the saveButton_onServerClick() code around the method call instead of using the events? It's because of what's supposed to happen if the row is not modified: you still want to return to the bookmarked row. In other words, if you had used the Expert-generated Save button, which simply calls *save*(), and put all the code in the *onSave* event, it would work fine—but only if the row was modified. The *onSave* event would process the reply or new message, and users would be returned to the bookmarked row.

But if the row was not modified, and all the Save button did was call *save*(), then nothing would happen. No events would be fired, including *onSave*. You would not go back to the bookmarked row or switch back to the Browse page. Therefore you must do some coding around the *save*() method call.

You could have put the new message processing in the *onSave* and done something like this:

```
function saveButton_onServerClick()
{
  if ( this.form.rowset.modified ) {
    if ( !this.form.rowset.save() ) {
      return;
    }
  }
  else {
    this.form.rowset.goto( this.form.bookmark );
  }
  this.form.refreshUnlinked();
  this.form.pageno = 1;
}
```

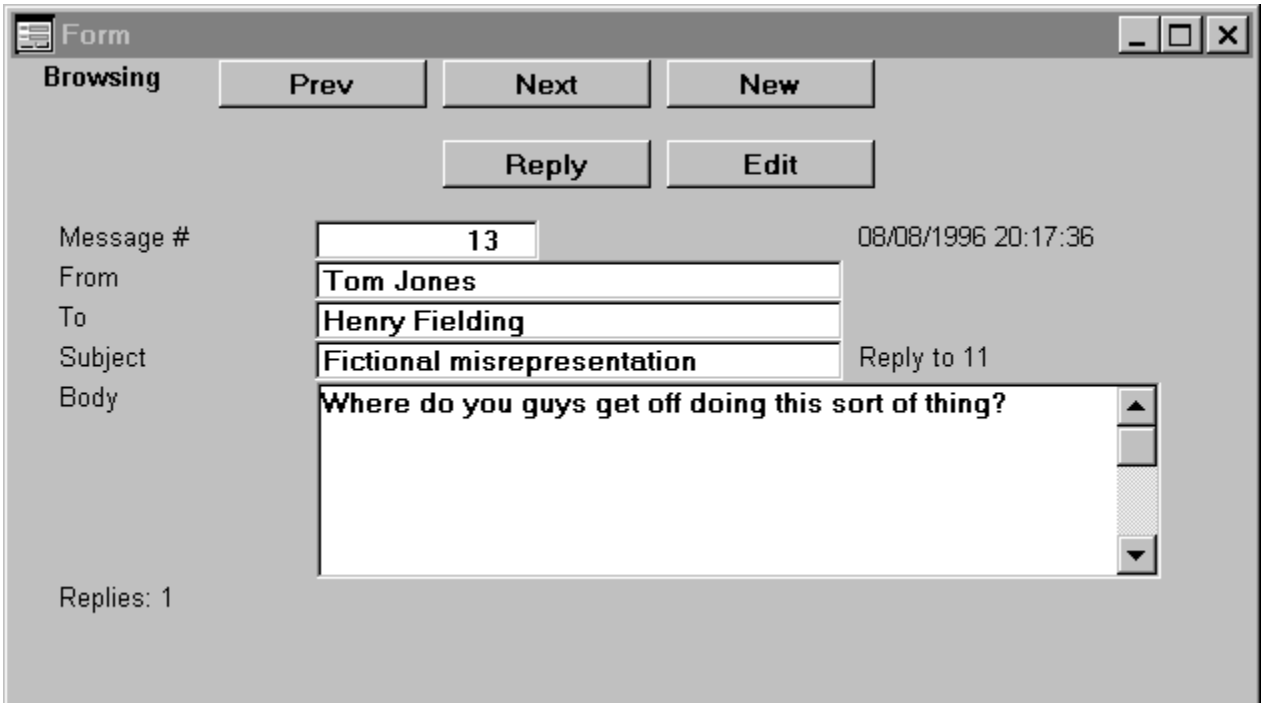
But this would separate the logic concerning when to go back to the bookmarked row into two different methods with no benefits. It's a bit more convenient to see all the logic in one place.

Because JavaScript programming is flexible, there is rarely just one right way to do something. You do need to know the differences and when they matter.

The TMD project: Testing the application

Related topics

Start the TMD from the Login form. Register yourself as a new user. Add a new message and save it. Reply to the message. Notice how the Reply to and # of replies fields are updated and displayed.



The TMD project: Consolidating code

Related topics

Here's an example using a method to consolidate common code called by other methods.

As you've seen, replying and adding a new message are closely related. In both cases users have to switch to the Editing page, page 2.

It would also be helpful if the Editing page displayed which type of message the user is composing.

To set up common code consolidation, follow these steps:

1. On page 2, add an HTML object below the editingLabel.
2. Set the HTML object's name to composeLabel.
3. Set the HTML object's *text* property to "Reply".
4. Create the following new method in the Method Editor:

```
function compose( cLabel )
{
  this.composeLabel.text = cLabel;     // Set compose label to "Reply" or
"New"
  this.rowset.refreshControls();
  this.rowset.modified = false;
  this.pageno = 2;                     // Switch to editing page
}
```

The appropriate word is passed as a parameter to the compose() method. After setting composeLabel, the form's controls are refreshed and the rowset's *modified* property is cleared (in case the row has been pre-filled). Then the form is switched to the Editing page.

To take advantage of this new method, modify the replyButton_onServerClick() method:

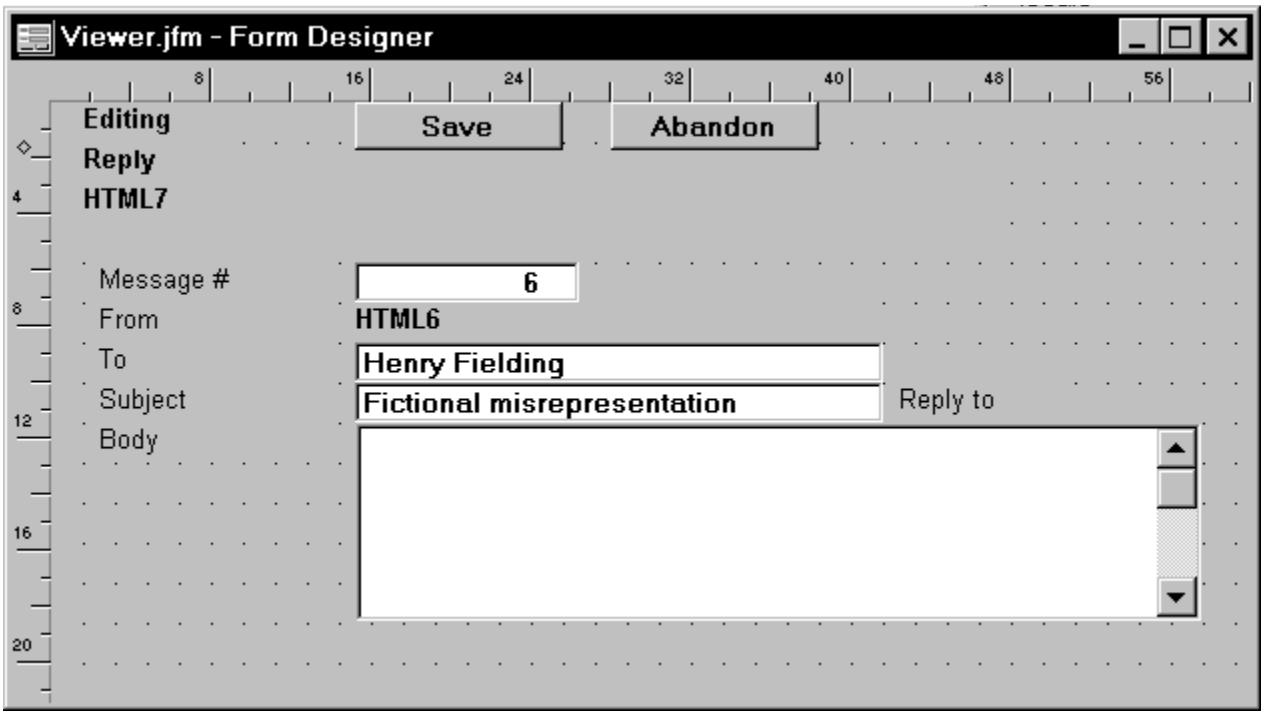
1. Add the highlighted lines, which also include support for the Thread root field, and ensure that the replyButton_onServerClick() method looks like this:

```
function replyButton_onServerClick()
{
  var nReplyTo = this.form.rowset.fields[ "Message #"   ].value;
  var cTo      = this.form.rowset.fields[ "From"        ].value;
  var cSubject = this.form.rowset.fields[ "Subject"     ].value;
  var nRoot    = this.form.rowset.fields[ "Thread root" ].value;
  this.form.rowset.beginAppend();
  this.form.rowset.fields[ "Reply to"    ].value = nReplyTo;
  this.form.rowset.fields[ "To"          ].value = cTo;
  this.form.rowset.fields[ "Subject"     ].value = cSubject;
  this.form.rowset.fields[ "Thread root" ].value = nRoot;
  this.form.compose( "Reply" );
}
```

2. Change the *onServerClick* event handler for newButton from the Expert-generated codeblock to this method:

```
function newButton_onServerClick()
{
  this.form.rowset.beginAppend();
  this.form.compose( "New" );
}
```

The result is tighter code, more reliable and easier to upgrade.



Viewer form with the *composeLabel*.

Conditionally displaying components

Related topics

So far the TMD project has stored a message number in the Reply To field when creating a reply, but there was no way to directly go to that parent message. We will now make it possible for the user to return to the parent message. Because this version of the TMD does not support message editing (once they are written they cannot be changed, so they can act as a permanent record), we will change the Edit button.

1. On page 1 in the Form Designer, set the Edit button's *name* to parentButton.
2. Set its *text* to "Go to parent".
3. Once you start navigating in the thread, you'll want an easy way to get back to where you started. So add another button.
4. Set the *name* of the second button to resumeButton.

Set the second button's *text* to "Resume". Set its *visible* property to false (it will still appear in the Form Designer).You don't want either of these buttons to be visible all the time. parentButton should be visible only if the message is a reply to another. Once parentButton is clicked, resumeButton should remain visible until it is clicked.

To implement this behavior, follow these steps:

1. Add the highlighted lines to the refreshUnlinked() method:

```
function refreshUnlinked()
{
  if ( this.rowset.endOfSet ) {
    this.replyToLabel.visible = false;
    this.numRepliesLabel.text = "";
    this.dateLabel.text       = "";
    this.parentButton.visible = false;
  }
  else {
    if ( this.rowset.fields[ "Reply to" ].value > 0 ) {
      this.replyToLabel.text    = "Reply to " +
                                  parseInt( this.rowset.fields[ "Reply
to" ].value );
      this.replyToLabel.visible = true;
    }
    else {
      // Thread root message, not a reply
      this.replyToLabel.visible = false;
    }
    this.numRepliesLabel.text = "Replies: " +
                                parseInt( this.rowset.fields[ "# of
replies" ].value );
    this.dateLabel.text       = this.rowset.fields[ "Posted" ].value;
    this.parentButton.visible = this.rowset.fields[ "Reply to" ].value > 0;
  }
}
```

2. parentButton saves a bookmark for the current row before navigating, so that resumeButton can get back to it. Type in the following *onServerClick* event handler for parentButton, overwriting the Expert-generated codeblock for the used by the Edit button:

```
function parentButton_onServerClick()
{
  if ( !this.form.resumeButton.visible ) {
    this.form.threadBookmark = this.form.rowset.bookmark();
    this.form.resumeButton.visible = true;
```

```
  }
  this.form.rowset.applyLocate( '"Message #" = ' +
       parseInt( this.form.rowset.fields[ "Reply to" ].value ) );
}
```

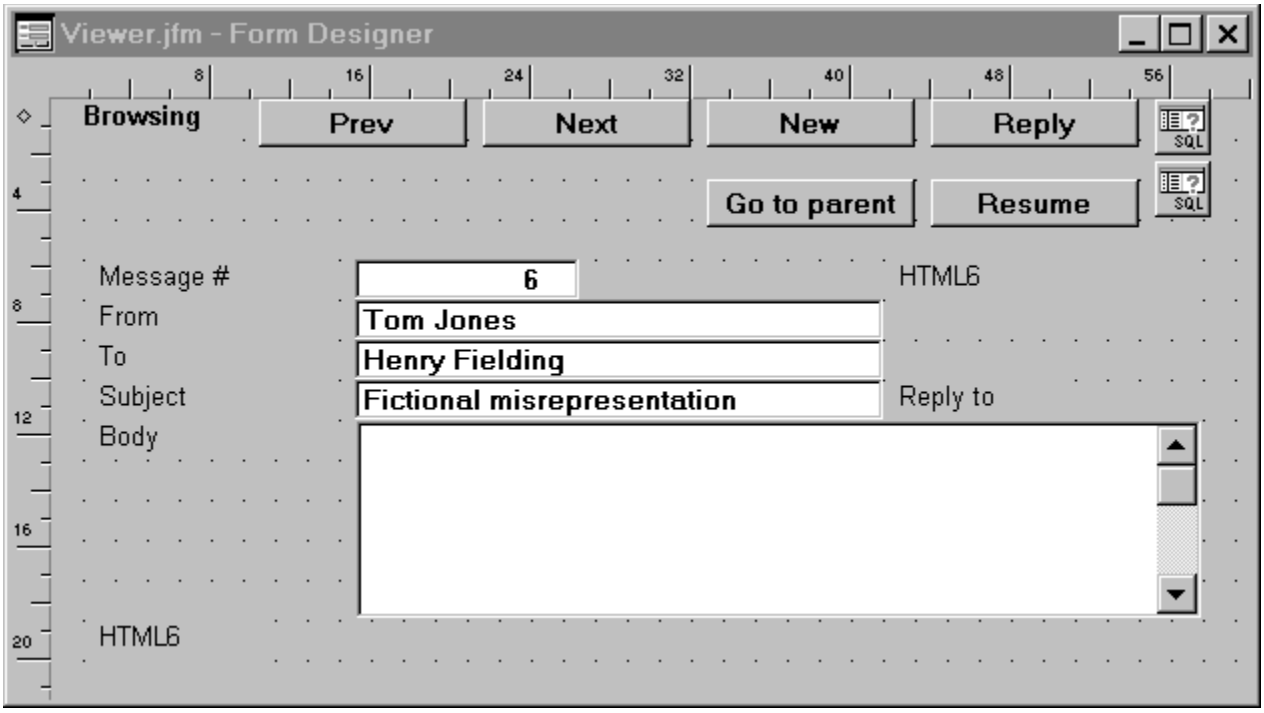
After saving the bookmark in the threadBookmark form variable, the method makes resumeButton visible. If it's already visible, then you don't want to do either, because that means the bookmark is already set. After that, the parent message is found by calling the *applyLocate*() method with a SQL expression.

3 Type in resumeButton's *onServerClick* event handler:

```
function resumeButton_onServerClick()
{
  this.form.rowset.goto( this.form.threadBookmark );
  this.visible = false;
}
```

This takes the user back to the bookmarked message and makes itself invisible. There's no need to check if threadBookmark has been set, because resumeButton would not be visible if it wasn't; and if it's not visible, you can't click it.

4 Save the form and try it out.



Message sections

Related topics

Sections are an extremely useful feature for organizing the TMD (or any application) according to topics, departments, or any kind of useful category. Each section can contain a separate grouping. For example, a company may have separate sections for each product. In the TMD, sections could organize message threads into different subject categories.

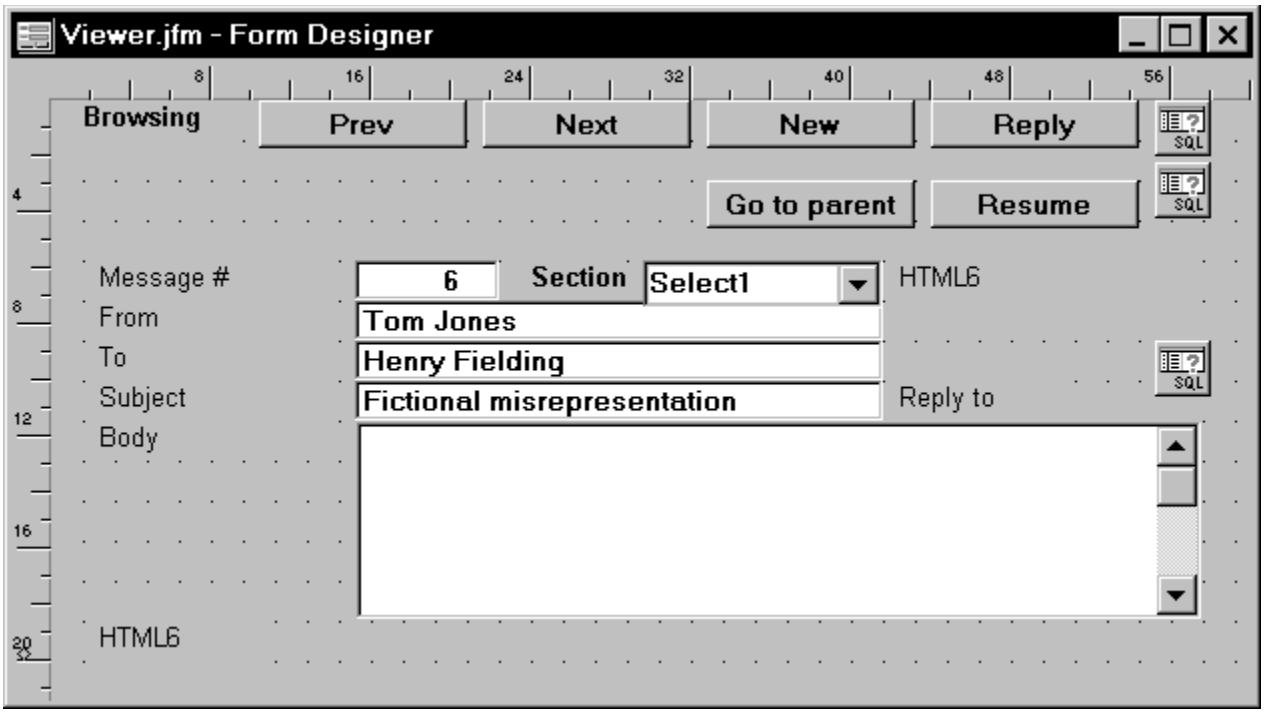
A section number will be stored in the table, while the name of the section will be displayed in a Select component on the form.

You need to populate the Sections table for testing purposes:

1. In the Tables tab of the IntraBuilder Explorer, double-click SECTIONS.DB to open the table for editing.
2. Add a few rows. For example, you might be creating a message database for pets:

Section #	Name
1	General
2	Dogs
3	Cats

3. Close the table form and open VIEWER.JFM in the Form Designer.
4. Drag the SECTIONS.DB table onto the form, which creates and activates sections1.
5. Switch to page 0.
6. To make some room in the top line, we want to reduce the size of the Message # Text component:
   1. Select the Message # Text component
   2. Set its template property to “999999”. This forces the Message # to display using six digits.
   3. Make the Text component narrower to fit the new size.
7. Drag an HTML component from the Component Palette onto the form and drop it to the right of the Message #.
8. Set its *text* to “Section”.
9. Drag a Select component from the Component Palette onto the form to the right of the Section label.Set the *name* to sectionSelect, and the *dataLink* to the Section field of the Messages query.


Viewer.jfm - Form Designer
Browsing
Prev
Next
New
Reply
Go to parent
Resume
SQL
SQL
Message #
6
Section
Select1
HTML6
From
Tom Jones
To
Henry Fielding
SQL
Subject
Fictional misrepresentation
Reply to
Body
HTML6

Populating an options array

Related topics

A Select component's *options* property dictates the options it displays. There are two types of *options*:

- Files in the current directory matching a specified file mask. For example, *.*, *.DB?, MESSAGES.*, and so on.
- An array.

The *options* Property Builder in the Form Designer allows you to specify a file mask or type in the elements of an array. This array would then be hard-coded into the form.

A more flexible approach would be to build the array from a table. This way, you can modify the *options* simply by editing the table, without having to touch your code.

Like the form, the Select component has an *onServerLoad* event, for which you should type the following method:

```
function sectionSelect_onServerLoad()
{
  this.aSections = new Array();
  this.form.sections1.rowset.first();
  while ( !this.form.sections1.rowset.endOfSet ) {
    this.aSections.add( this.form.sections1.rowset.fields[ "Name" ].value );
    this.form.sections1.rowset.next();
  }
  this.options = "array this.aSections";
}
```

The aSections array is created as a property of the component. Then a loop is used to navigate through the Sections table and populate the array. Finally, the contents of the array are copied into the Select component by assigning a string containing the word "array" plus a reference to the array to the component's *options* property.

Once the array elements have been copied into the component by assigning the *options* property, the array is not actually needed. You could use a plain variable for the array reference instead of a property of the component. By using a property, however, you can easily modify the array and update the component's *options*.

Two-way field morphing

Related topics

Now that the Select component displays a list of the available sections, you'll need to morph the section names into the section numbers stored in the Messages table, and the reverse, morphing the section numbers into names.

For numbers into names, or rather what is stored in the table into what is displayed, set the Section field's *beforeGetValue* event:

```
function messages1_section_beforeGetValue()
{
  if ( this.parent.parent.endOfSet ) {
    // When navigating to end-of-set
    return null;
  }
  else if ( this.value == null ) {
    // For beginAppend()
    return "";
  }
  else {
    // Normal lookup, with value in case lookup fails
    var r = this.parent.parent.parent.parent.sections1.rowset;
    return r.applyLocate( '"Section #" = ' + parseInt( this.value ) ) ?
           r.fields[ "Name" ].value : "Closed section";
  }
}
```

This *beforeGetValue* event handler is almost identical to the one for the From field. It looks up the Section # in the Sections table through sections1 and returns the corresponding name, or "Closed section" if no match is found. But because "Closed section" is not in sectionSelect's *options* array, it would never be displayed.

Field morphing in the other direction, converting what is assigned to a component's *value* property into what is assigned to the field, is a bit more complicated. It uses the field's *canChange* method. *canChange* fires when attempting to assign something to a field's *value* property. Like other can- events, it returns true or false to indicate whether the assignment actually occurs.

To implement field morphing,

1. Assign the morphed value in the *canChange* event handler and return false so that the component's *value* does not get written.
2. Type in the following *canChange* event handler for the Section field:

```
function messages1_section_canChange( newValue )
{
  var r = this.parent.parent.parent.parent.sections1.rowset;
  if ( r.applyLocate( '"Name" = \'' + newValue +'\'' ) ) {
    this.value = r.fields[ "Section #" ].value;
  }
  return false;
}
```

When *canChange* is fired, the proposed new value is passed as a parameter. In this particular method, it is assigned to the variable newValue, but you can name this parameter anything you want.

Then the method tries to match newValue with names in the Sections table with the *applyLocate*() method. A match should be found, because the possible values are the same ones that were read from the Sections table that populated the sectionSelect object's *options* array.

- If a match is found, the corresponding Section # is stored in the *value* property of the Section field in the Messages table.
- If there is no match, nothing happens.

Either way, the method returns false, so only valid section numbers are written to the Section field.

The TMD project: Setting default section numbers

Related topics

With the Section field now limited to a strict set of options by its field morphing events, you should make sure that all new rows get assigned a valid default value.

For replies, copy the section by adding the highlighted lines to the replyButton_onServerClick() method:

```
function replyButton_onServerClick()
{
  var nReplyTo = this.form.rowset.fields[ "Message #"   ].value;
  var cTo      = this.form.rowset.fields[ "From"        ].value;
  var cSubject = this.form.rowset.fields[ "Subject"     ].value;
  var nRoot    = this.form.rowset.fields[ "Thread root" ].value;
  var cSection = this.form.rowset.fields[ "Section"     ].value;
  this.form.rowset.beginAppend();
  this.form.rowset.fields[ "Reply to"    ].value = nReplyTo;
  this.form.rowset.fields[ "To"          ].value = cTo;
  this.form.rowset.fields[ "Subject"     ].value = cSubject;
  this.form.rowset.fields[ "Thread root" ].value = nRoot;
  this.form.rowset.fields[ "Section"     ].value = cSection;
  this.form.compose( "Reply" );
}
```

For new messages, pre-fill the field. Add the highlighted line to the newButton_onServerClick() method:

```
function newButton_onServerClick()
{
  this.form.rowset.beginAppend();
  this.form.rowset.fields[ "Section" ].value =
this.form.sectionSelect.aSections[ 0 ];
  this.form.compose( "New" );
}
```

This assigns the first option in the generated array of sections to the Section field as the default. It was fortunate you kept that array as a property of the form.

Controlling the rowset's sort order

Related topics

The Section field wouldn't be very useful if the messages were not sorted by section. There are two ways to do this:

- Specify fields in the ORDER BY clause in the SQL SELECT command in the query's *sql* property.
- Create and use an index. This option is only available for Standard tables.

An index is more flexible; you can simply set the rowset's *indexName* property whenever you want and the rowset will change its order. For SQL, you would have to deactivate the query, change the *sql* property, and reactivate.

Creating indexes

Related topics

Indexes are created and managed in the Table Designer. To create an index,

1. Open the table.
2. Choose Structure|Manage Indexes.
3. In the Manage Indexes dialog box, choose New or Edit. This displays a different Define Indexes dialog box, depending on which Standard table type you're using.

DBF tables

Related topics

DBF tables use expression indexes. Instead of a list of fields, DBF indexes are defined as expressions that evaluate into a simple data value; either a character string, number, date, or logical value. These expressions may use basic dBASE functions. Although IntraBuilder does not understand dBASE functions, the Borland Database Engine does, so you can create and use DBF indexes in IntraBuilder.

For a single field, the index expression consists simply of that field's name. For multiple fields, they are concatenated to form a single value that cannot exceed 100 bytes. Multiple character fields are simply added together as they are in JavaScript, but to add fields of different types, they must be converted with dBASE functions.

When different field types are combined, they are almost always all converted to character. If you use the Field List in the Define Index dialog box, IntraBuilder will automatically include the necessary conversion functions and concatenate the fields.

DBF indexes also support a For expression including only those rows that match the expression, and filter out the rest. This expression may use dBASE functions.

You can create the index so that it includes only the first occurrence of each key value in the table. This is not the same as a primary key, which forbids multiple rows with the same key value. With DBFs, the option simply ignores subsequent rows with the same key value.

DB tables

Related topics

DB tables use a list of fields. To create indexes in a DB table, it must have a primary key. When you choose the Manage Indexes option from the menu, the Primary Key dialog box will appear first (if the table does not have a primary key).

Common index attributes

Related topics

Both DBF and DB indexes may be ascending or descending. Each index must have a name that follows the same rules as each table format's field names.

The TMD project: Creating the Thread index

Related topics

To create the thread index,

1. Open the MESSAGES.DB table in the Table Designer.
2. Choose Structure|Manage Indexes from the menu.
3. Choose New from the Manage Indexes dialog box.
4. Create a new index based on the fields Section, Thread root, and Message #. The index should be ascending.
5. Set the name of the index to “Thread.”
6. Click OK for both the Define Index dialog box and the Manage Indexes dialog box.
7. Close the Table Designer.

This is the result: This index will display the messages by section. Within each section, messages will be grouped by thread, with the oldest threads first. Within each thread, messages will be in chronological order.

Setting the indexName property

Related topics

To set the indexName property,

1. Open the VIEWER.JFM form in the Form Designer.
2. Set message1's *rowset*'s *indexName* property to "Thread".
3. Save and run the form.

The messages will now be displayed in thread order.

Note If you test the TMD after setting the *indexName*, the messages may appear in a strange order, with replies appearing before the original message. This is because the Thread root field was not filled in during The TMD project: Testing the application; that support was added later, in The TMD project: Consolidating code.

You should empty the Messages table again and add all-new messages, now that both the Thread root and Section support is enabled.

High Message Number

Related topics

A message database would quickly become unusable if it did not track the messages you've read in some way. Otherwise it would up to you to sift through all the old messages to find the new ones.

There are a number of strategies for tracking which messages have been read. They depend on how the message database is implemented.

For a database that uniquely numbers each message in straight chronological order, one of the simplest and most reliable ways is to keep track of a user's High Message Number (HMN). The HMN is the highest number message that the user has read. Every time the user checks the message database, all messages with that number and below are filtered out. This leaves all the messages posted after last one the user read, if any.

The TMD project: Tracking the HMN

Related topics

The Users table has an HMN field to track each user's High Message Number. When the user logs in, the Messages table is checked to see if there are any messages with a higher number; that is, whether there are any new messages for that user.

If there are no new messages, the user should be so informed. At this point, users would be given the opportunity to use an administration feature to set their HMN to a lower number—in order to view previously read messages.

If there are new messages, the Messages table is then checked for messages specifically addressed to the user. The user is informed that there are new messages, whether or not there are any messages addressed to them, and given two options: to go into the message viewer or use the administration feature to change their HMN.

Because there is no formal concept of leaving the message viewer (you can simply point your browser somewhere else), the value of the user's HMN field is updated as he or she reads each message.

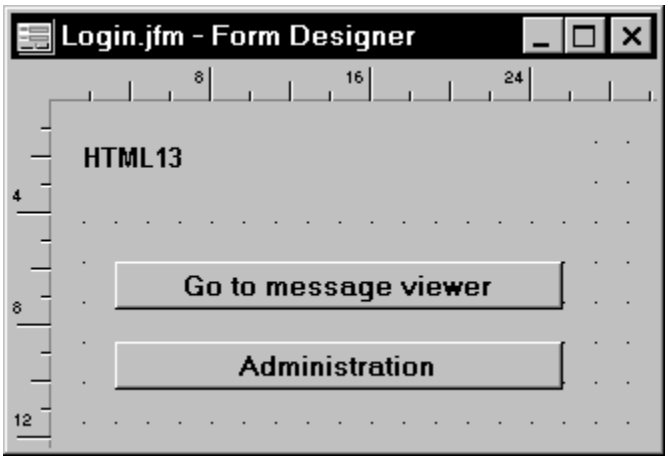
Checking for new messages

Related topics

Follow these steps to check for new messages when the user logs in:

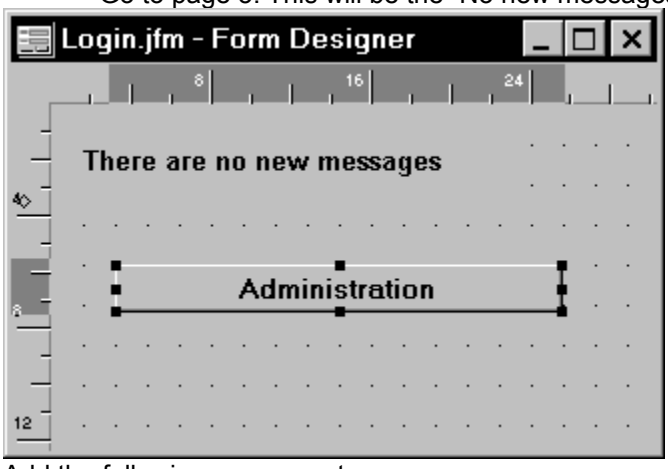
1 Open LOGIN.JFM in the Form Designer.

2 To check the user's HMN against the Messages table, the table must be open. Drag the MESSAGES.DB table from the Explorer onto the form to create the query messages1. Set its *locateOptions* property to Ignore case.
Go to page 4. This will be the "There are new messages" page.



Add the following components:

- An HTML component. Set its *name* to waitingLabel.
- A Button component. Set its *text* to "Goto message viewer" and its *name* to viewerButton.
- A Button component. Set its *text* to "Administration" and its *name* to adminButton.
- Go to page 5. This will be the "No new messages" page.



Add the following components:

- An HTML component with the *text* "There are no new messages".
- A Button component with the *text* "Administration" *name*ed adminButton2.

3 Create a new method named checkNewMessages():

```
function checkNewMessages()
{
  // See if there are any new messages
  this.messages1.rowset.filter = '"Message #" > ' +
                                 parseInt( this.rowset.fields[ "HMN" ].valu
e );
  if ( this.messages1.rowset.endOfSet ) {
```

```
      // No new messages
      this.pageno = 5;
    }
    else {
      this.pageno = 4;
      // Check for waiting messages
      this.waitingLabel.text = this.messages1.rowset.applyLocate( '"To" = \''
+
                               this.rowset.fields[ "User name" ].value +
'\'') ?
                               "You have waiting messages!" : "No waiting
messages";
    }
  }
```

4 Change the loginButton_onServerClick() and tryNameButton_onServerClick() methods so that they call checkNewMessages() instead of running VIEWER.JFM directly:

```
  function loginButton_onServerClick()
  {
    if ( this.form.rowset.applyLocate() ) {
    ~~_sys.forms.run( "VIEWER", this.form );~~ remove these lines
    ~~this.form.close();~~                     and replace them with
      this.form.checkNewMessages();
    }
    else {
      this.text = "Try again";
      this.form.beginLogin();
    }
  }
  function tryNameButton_onServerClick()
  {
    try {
      var e = new Exception();  // Create Exception object in case you need
to throw
      if ( this.form.users2.rowset.applyLocate( '"User name" = \'' +
           this.form.userName1.value + '\'' ) )
      { // If user name is already used
        e.message = "Login name already in use";
        throw e;  // set the error message and throw the exception
      }
      if ( this.form.password1.value != this.form.password2.value ) {
        e.message = "Passwords do not match";
        throw e;  // Same if passwords don't match
      }
      // If you get this far, there were no errors
      this.form.rowset.save();  // so save the new user
    ~~_sys.forms.run( "VIEWER", this.form );  // and login~~  remove these
lines
    ~~this.form.close();~~                                  and replace them
with
      this.form.checkNewMessages();
    }
    catch ( Exception e ) {     // When there's an exception
      this.text = e.message;    // Set the button's text to the appropriate
message
    }
  }
```

5 Create the following *onServerClick* event handler for viewerButton:

```
function viewerButton_onServerClick()
{
  _sys.forms.run( "VIEWER", this.form );
  this.form.close();
}
```

6 Save the form to be on the safe side, but don't yet close the Form Designer.

checkNewMessages() works by first setting a *filter* in the Messages table to filter out all the old messages, those with a message number lower than or equal to the user's HMN. If the rowset is at the end-of-set after setting the filter, this means that there are no new messages, so the "No new messages" page is displayed.

If the rowset is not at the end-of-set, then the "There are new messages" page is displayed. *applyLocate*() is called to look for messages addressed specifically to the user, with the *filter* still in effect. This means that *applyLocate*() will find only new messages addressed to the user. These are considered waiting messages. The *text* of the waitingLabel component is set accordingly.

When the user clicks the viewerButton it starts the message viewer form and closes the login form, as it did before.

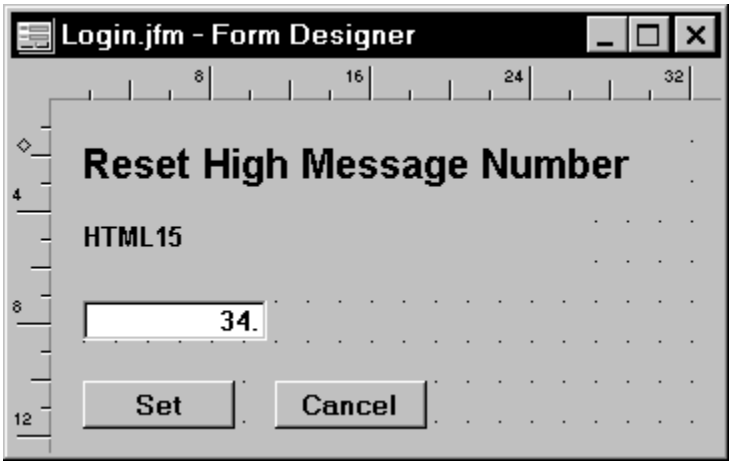
Clicking on either adminButton or adminButton2 doesn't do anything—yet.

The TMD project: Administering the HMN

Related topics

Follow these steps to add HMN administration:

1 Go to page 6. This will be the “Reset HMN” page.



Add the following components:

- An HTML component with the text “<H2>Reset High Message Number</H2>”.
- An HTML component with *name* set to msgRangeLabel.
- From the Field Palette, add the HMN field from users1 query. Its *name* is automatically set to hmn.
- A Button component with the *text* set to “Set” and the *name* set to setHMNButton.
- A Button component with the *text* set to “Cancel” and the *name* set to cancelHMNButton.

2 Create the method resetHMN():

```
function resetHMN()
{
  this.messages1.rowset.clearFilter();
  this.messages1.rowset.first();
  this.minHMN = parseInt( this.messages1.rowset.fields[ "Message #" ].value
);
  this.messages1.rowset.last();
  this.maxHMN = parseInt( this.messages1.rowset.fields[ "Message #" ].value
);
   this.msgRangeLabel.text = "Messages in the database are numbered from " +
                             this.minHMN + " to " + this.maxHMN
  this.pageno = 6;
  this.rowset.beginEdit();
}
```

3 Create the following *onServerClick* event handler for setHMNButton:

```
function setHMNButton_onServerClick()
{
    if ( this.form.hmn.value >= 0 &&
         this.form.hmn.value <= this.form.maxHMN ) {
    this.form.rowset.save();
    this.form.checkNewMessages();
  }
  else {
    this.form.msgRangeLabel.color = "red";
  }
}
```

4 Create the following *onServerClick* event handler for cancelHMNButton:

```
function cancelHMNButton_onServerClick()
{
  this.form.rowset.abandon();
  this.form.checkNewMessages();
}
```

5 Create the following *onServerClick* event handler for adminButton:

```
function adminButton_onServerClick()
{
  this.form.resetHMN();
}
```

6 Link the adminButton_onServerClick() method to adminButton2's *onServerClick* event as well.

7 Go to page 1, save the form, and close the Form Designer.

Both the administration buttons run the form's resetHMN() method. resetHMN( ) starts by clearing any filter on the Messages table and gets the lowest and highest message numbers in the table. These are stored as form variables.

A *text* of the msgRangeLabel component is set to inform the user of the message number range. The form is switched to page 6 and the User rowset is switched to Edit mode.

The hmn component from the Field Palette is *dataLink*ed to the user's HMN field, so it automatically displays the current value of the field and will update the field if a new value is saved.

If the user clicks on the Set button, the new HMN *value* will be checked to make sure it falls in the acceptable range. The new HMN cannot higher than the current highest message number, but can be as low as zero. It never hurts to set the HMN below the lowest numbered message in the table; it simply means that all the messages in the table are considered new.

If the new HMN is acceptable, then the changes are saved and the form's checkNewMessages() method is run again, which handles the job of switching back to the appropriate page to indicate whether there are new messages for the user. If the new HMN is not acceptable, the message range is changed to red and the user is given another chance.

If the user clicks the Cancel button, any changes to the rowset are abandoned and the form's checkNewMessages() method is run again. This handles the job of switching back to the appropriate page to indicate whether there are new messages for the user.

The TMD project: Using the HMN in the message viewer

Related topics

Now with the HMN fully supported on the login side, the viewer must also be augmented.

Borrowing data access objects from other forms

Related topics

The Viewer form's *onServerLoad* event handler looks for a form reference as a parameter and uses that form reference to access the row in the Users table that describes the user that has logged in to the message viewer. At present, it just reads the fields from the table and stores the values as form variables.

```
this.userID   = VIEWER.arguments[ 0 ].rowset.fields[ "User ID"   ].value;
this.userName = VIEWER.arguments[ 0 ].rowset.fields[ "User Name" ].value;
```

Now that the HMN needs to be updated every time the user reads a new message, you must be able to access that row in the Users table. There are a number of ways to handle it:

- There's already a query with the Users table that's used as a lookup table for the From field's *beforeGetValue* event (added in One-way field morphing). Because you have the user ID, you could do an *applyLocate*() when needed and update the user's HMN field.
- Instead doing an *applyLocate*() over and over, you could find the matching row first and set a *bookmark*() to it in a form variable. Then you simply *goto*() the bookmark.
- You could create another Query object in the Viewer form just for HMN updates. After using the user ID to locate the matching row, you can simply leave the rowset positioned at that row.
- The Viewer form could borrow the Query object in the Login form.

Borrowing is the easiest solution. The rowset in the query in the Login form is already positioned at the correct row. By saving a reference to the Login form's Query object as a property of the Viewer form, the Query object will not be destroyed when the Login form is closed. You can continue to access its rowset to update the user's HMN.

Filtering rows

Related topics

To actually use the HMN, you want to able to restrict the rows shown to those messages that come after it. But you also want to be able to temporarily show older messages if the user starts threading to parent messages.

There are three mechanisms that can be used to filter out rows so that you see only those that match a specified condition.

SQL WHERE clause

Related topics

In a query's *sql* property, the SQL SELECT statement can contain a WHERE clause. The WHERE clause specifies a SQL expression. Because the filter is in the SQL SELECT, the database engine can optimize it so that the resulting rowset is returned quickly and can be easily navigated. On the flip side, because the filter condition is actually part of the SQL SELECT, you cannot easily turn it on and off.

canGetRow event

Related topics

The rowset has a *canGetRow* event that fires every time it attempts to fetch a row. This event returns true or false to indicate whether a particular row can be fetched. Thus, in effect, you can apply a filter by using any JavaScript code you want.

Inside the *canGetRow* event handler, *this* refers to the row being fetched; in other words, inside *canGetRow* the row has actually already been obtained, and the event handler's return value indicates whether you can keep the row or tell the database engine to get another one.

The main drawback of *canGetRow* is that because it's JavaScript code, the condition is not optimizable by the database engine. IntraBuilder must actively call *canGetRow* every time it navigates for each row it encounters until it finds a match or hits the end-of-set. Therefore *canGetRow* should not be used to filter out large blocks of rows— which is exactly what the HMN filter is. *canGetRow* works best when filtering out a minority subset of rows that is evenly dispersed through the rowset.

filter property

Related topics

The best solution for the HMN feature is to use a rowset's *filter* property, either by assigning a SQL expression directly to the property or using *beginFilter*(). Because the *filter* property is a SQL expression, it is optimizable. You can also easily remove the filter temporarily.

Because the *filter* is used to set a global condition—only messages after the HMN—you can't use Filter-By-Form at the same time, for example to search for a particular subject. There is only one *filter*, so using *beginFilter*( ) would overwrite the HMN filter condition. Although you could construct a *filter* expression that combines the HMN restriction and any search criteria, that would require manual coding, which defeats the purpose of using the Filter-By-Form feature.

Searches are still possible with Locate mode. You can even simulate Filter-By-Form by using a combination of Locate-By-Form and *locateNext*().

The TMD project: Applying the HMN filter

Related topics

To bring everything together, you'll need to make some minor modifications and additions:

1 Remove the two struck lines and add the highlighted lines so that the Form_onServerLoad() event looks like:

```
function Form_onServerLoad()
{
  try {
    ~~this.userID   = VIEWER.arguments[ 0 ].rowset.fields[ "User ID" ].value;~~  **// REMOVE**
    ~~this.userName = VIEWER.arguments[ 0 ].rowset.fields[ "User Name" ].value;~~  **// REMOVE**
    **this.userRowset = VIEWER.arguments[ 0 ].rowset;**
    **this.userQuery  = this.userRowset.parent;**
    **this.userID     = this.userRowset.fields[ "User ID" ].value;**
    **this.userName   = this.userRowset.fields[ "User name" ].value;**
    **this.HMN        = this.userRowset.fields[ "HMN" ].value;**
  }
  catch ( Exception e ) {
    this.userID   = 0 ;              // Eventually will not allow access
    this.userName = "Unregistered" ; // But for now allow unregistered user
    **this.HMN      = 0;**
  }
  // Set "From" name on compose page to user name, because it will never change
  this.userLabel.text = this.userName;
  **this.applyHMN();**
  this.refreshUnlinked();
}
```

The new code stores a reference to the rowset in the login form as the form variable userRowset, and reference to the query itself (the rowset's parent) as userQuery. The rowset reference is the one that is actually used, because the rowset has all the relevant property and methods. But a reference to the query must be stored so that the Query object and its rowset are not destroyed when the Login form closes.

The user ID and name are extracted with a variation of the previous code. The HMN is stored as well. If the user did not open the Viewer through the Login form, the HMN is set to zero.

Just before calling refreshUnlinked(), applyHMN() is called to set the filter.

2 Create a new form method named applyHMN():

```
function applyHMN()
{
  this.rowset.filter = '"Message #" > ' + parseInt( this.HMN );
}
```

3 Add the highlighted line to parentButton_onServerClick() to disable the HMN filter when looking for the parent message:

```
function parentButton_onServerClick()
{
  if ( !this.form.resumeButton.visible ) {
    this.form.threadBookmark = this.form.rowset.bookmark();
    this.form.resumeButton.visible = true;
    **this.form.rowset.clearFilter();**
  }
  this.form.rowset.applyLocate( '"Message #" = ' +
```

```
        parseInt( this.form.rowset.fields[ "Reply to" ].value ) );
  }
```

4 You'll want to restore the filter when users stop threading, so add the highlighted line to resumeButton_onServerClick()

```
  function resumeButton_onServerClick()
  {
    this.form.applyHMN();
    this.form.rowset.goto( this.form.threadBookmark );
    this.visible = false;
  }
```

5 You need to update users' HMN field as they navigate through the Messages table. Add the highlighted line to messages1_onNavigate():

```
  function messages1_onNavigate()
  {
    this.parent.parent.refreshUnlinked();
    this.parent.parent.updateHMN();
  }
```

6 Finally, create the updateHMN() method:

```
  function updateHMN()
  {
    if ( !this.rowset.endOfSet ) {
      if ( this.userID != 0 ) {  // Cannot update HMN for unregistered user
        if ( this.rowset.fields[ "Message #" ].value >
             this.userRowset.fields[ "HMN" ].value ) {
          this.userRowset.fields[ "HMN" ].value =
this.rowset.fields[ "Message #" ].value;
          this.userRowset.save();
        }
      }
    }
  }
```

Now save the Viewer form and run the Login form to give the HMN a whirl.

Project summary

Related topics

The Threaded Message Database has come a long way. In this series of topics you've added a few important enhancements for a robust real-world intranet application:

- By requiring users to login, you've learned how to construct custom login forms that can be used to greatly enhance an enterprise intranet system.
- You've programmed the TMD to redisplay the original message to which a user has just sent a reply.
- Now the TMD lets users navigate in the message thread and resume where they left off.
- You've added sections to classify messages any way you want. Now TMD sorts messages are sorted in thread order, grouped by subject.
- The TMD tracks the user's High Message Number (HMN) so that users need only browse new, unread messages but can reset the HMN to view previously read messages.

Custom forms and components

Related topics

One of the goals of object-oriented programming is reusability: Design something once and use it over and over.

Custom forms and components exemplify reusability. Custom forms allow you to design a common look-and-feel for your forms, that can be easily changed. You can design palette-fulls of components that you can drop onto your forms and reports. You can share custom components with others to multiply your productivity.

Custom classes also make your code more granular. Instead of storing all the properties and methods in the *class* definition for the form, the properties and methods for each custom component can be stored in that component's own *class* definition, reducing the clutter in the form definition, and making your applications easier to maintain.

Custom forms

Related topics

Custom forms are used to codify attributes shared by multiple forms, including:

- A common look for your forms, such as a company logo on every form or a consistent background color key or an official bitmap;
- Common components, such as navigation controls and icons that always take you back to a home page;
- Common behaviors, such as a method that performs a calculation.

You define a custom form class that contains those attributes. Then forms derived from that custom form class inherit its attributes.

Defining a custom form class

Related topics

There are three ways to define a custom form class:

- Save a form as a custom form class in the Form Designer.
- Use the Custom Form Class Designer.
- Write the JavaScript code from scratch.

Custom forms are stored in JCF (JavaScript Custom Form) files. A JCF file is similar to a JFM file. It contains one or more *class* definitions that describe a form. A JCF file does not contain a Header section.

The Form Designer and Custom Form Class Designer are detailed in Form Designer. Of course, you can also manually create a JCF file with the Script Editor.

The TMD project: Creating a common look-and-feel

Related topics

In a modest attempt to spruce up the TMD forms and visually unify them, we will establish a more cheerful color than the default silver/gray background. The new color scheme will be established in a base form so that it is reproduced in all the derived forms.

To create this new base form,

1. Start the Custom Form Class Designer by double-clicking the second (Untitled) icon in the Forms page of the IntraBuilder Explorer. (The Custom Form icon appears more empty than the fuller-looking Form icon.)
2. Right-click the form and choose Inspector from context menu.
3. Go to the Properties page of the Inspector and set the *color* property (under Visual Properties in the Inspector outline) to

   ```
   lemonchiffon
   ```

   Type this name directly into the Inspector and press Enter.
4. Close the Custom Form Class Designer. Because the form has not been saved, you will be prompted for a file name. Type tmdbase and press Enter. This creates the file TMDBASE.JCF.

Examining the custom form definition

Related topics

In Forms tab of the IntraBuilder Explorer, right-click TMDBASE.JCF. Choose Edit as Script from the context menu. The JavaScript code should look like this:

```
class tmdbaseCForm extends Form custom {
   with (this) {
      color = "lemonchiffon";
      height = 20;
      left = 36.5;
      top = 0;
      width = 60;
      title = "Form";
   }
}
```

This code is similar to code generated for a normal form definition. The two notable differences are:

- In automatically naming the form class, “CForm” is added to the file name instead of just “Form”. This is simply the default naming convention used by the Custom Form Class Designer. If you had used the Save As Custom dialog box as detailed in Form Designer, you could use any name you want.
- The reserved word *custom* is used at the end of the *class* statement. This identifies the class as a custom class, which is needed when creating, in the Form Designer, a new form based on the custom class. It informs the Form Designer that any properties or components defined by the custom class do not need to be streamed out in the resulting JFM file, because they are already defined in the JCF file that contains the custom form class upon which the form is based.

If you want to write your own custom form class from scratch, be sure to include the *custom* reserved word.

Assigning a custom form class to a form

Related topics

To use a custom form class, you can choose the Set Custom Form Class menu option in the Form Designer. Once a custom form class has been assigned as the form's base class, that form will continue to use that base class until it is set to something else or cleared.

Note Whenever you use the Set Custom Form Class menu option in the Form Designer, the Form Designer continues to use that base form for all new forms that are created—existing forms continue to use whatever base form, if any, they have been assigned—until the base form is cleared. This can be inconvenient if you're creating multiple forms that aren't all using that base form; you would have to repeatedl;y change or clear the base form in the Set Custom Form Class menu option.

To avoid being saddled with a base form class, you can bypass the Set Custom Form Class menu option and manually set the base form class in the JFM file. This is what the Form Designer does for you through the menu option.

For example, the class definition for the Login form starts with the line

```
class loginForm extends Form {
```

If you use the Set Custom Form Class menu option to set the form's custom form class to the tmdbaseCForm class in the file TMDBASE.JCF, the line becomes

```
class loginForm extends tmdbaseCForm from "TMDBASE.JCF" {
```

Instead of being derived directly from the stock Form class, the form is derived from the tmdbaseCForm class, which in turn is derived from the Form class. The reserved word *from* is used to designate the script file that contains the base class definition.

The TMD project: Assigning the custom form class

Related topics

Now that you've created a base form class, you need to assign it to the existing Login and Viewer forms:

1. In the Forms page of the IntraBuilder Explorer, right-click LOGIN.JFM and choose Edit as Script from the context menu.
2. Change the *class* statement at the top of the file from

   ```
   class loginForm extends Form {
   ```

   to

   ```
   class loginForm extends tmdbaseCForm from "TMDBASE.JCF" {
   ```
3. Save the form and close the Form Designer.
4. In the Forms page of the IntraBuilder Explorer, right-click VIEWER.JFM and choose Edit as Script from the context menu.
5. Change the *class* statement at the top of the file from

   ```
   class viewerForm extends Form {
   ```

   to

   ```
   class viewerForm extends tmdbaseCForm from "TMDBASE.JCF" {
   ```
6. Save the form and close the Form Designer.

Now run the TMD application and bask in the glow of the new color scheme. (The lemonchiffon color may be dithered in IntraBuilder, but should appear as a solid color when run over a browser.)

Custom visual components

Related topics

The idea behind custom components is to preset everything you need.

A *Select* component that has the abbreviations for the fifty states in the USA is a good example. You could build such a component whenever you want in the Form Designer. With the Visual Array Builder, the process is simple, but tedious and fraught with danger; the fifty states of the USA are considerably harder to remember than the seven dwarves. And what if they add a state, or lose one? With a custom component, you only have to change the *options* array once. Otherwise, you would have to separately change every single state Select component you have.

In addition to tangible properties like *options* and *color*, custom components also have preset methods and event handlers. Because the definition of a custom component is in a CC file, it is not touched by the Form Designer, so you can put whatever code you want in its constructor.

Creating custom components

Related topics

There are two ways to create a custom component:

- Save any control on a form as a custom component in the Form Designer.
- Write the JavaScript code from scratch.

Custom components are stored in CC (Custom Component) files. A CC file is like a JCF file, in that it contains one or more *class* definitions. In a CC file, these classes are derived from components like Button and Select, instead of Form.

Creating custom components with the Form Designer is detailed in Form Designer. You can create a JCF file with the Script Editor.

The TMD project: Using custom components

Related topics

In this version of the TMD project, there are no components used in multiple forms, so there is no immediate benefit in using custom components. But the section Select component is fairly generic, in that it reads its *options* from a field in a table.

In order for a component to be truly reusable, it must be designed flexibly enough to be dropped into a myriad of situations. From the section Select object you can make a table-driven Select component that you can reuse in all your projects, as well as the TMD.

The first step is to save the existing section Select component as a custom component:

1. Open VIEWER.JFM in the Form Designer.
2. Click the sectionSelect component.
3. Choose File|Save as Custom... from the menu.
4. In the Save as Custom dialog box, make sure that Save Select Components as Custom is selected.
5. Set the Class Name to “FieldSelect”.
6. Set the Custom Component File to “SELECT.CC”.
7. Click OK.
8. Close the Form Designer.

Examining the custom component definition

Related topics

In the Custom tab of the IntraBuilder Explorer, right-click SELECT.CC. Choose Edit as Script from the context menu. The JavaScript code should look like this:

```
class FieldSelect(FormObj) extends Select(FormObj) custom {
   with (this) {
     left = 37;
     top = 4;
     width = 15;
     dataLink = parent.messages1.rowset.fields["Section"];
     pageno = 0;
     onServerLoad = class::sectionSelect_onServerLoad;
   }
   function sectionSelect_onServerLoad()
   {
     this.aSections = new Array();
     this.form.sections1.rowset.first();
     while ( !this.form.sections1.rowset.endOfSet ) {

this.aSections.add( this.form.sections1.rowset.fields[ "Name" ].value );
       this.form.sections1.rowset.next();
     }
     this.options = "array this.aSections";
   }
}
```

The *class* definition contains both the property settings for the component, and any methods assigned to it.

Custom component class declaration

Related topics

The *class* line indicates that the FieldSelect class is derived from the stock Select class. As with custom form classes, the reserved word *custom* is used to identify this class as a custom component.

Like all visual component classes, the Select class requires a parameter that identifies the Form object to which the Select object binds itself. The Form object is represented by the reserved word *this* in the constructor for the form, as shown in the first line of this typical *with* statement.

```
with ( this.someSelect = new Select( this ) ){
```

When a stock component class is made into a subclass, the subclass must pass along that parameter. Therefore in the definition of the FieldSelect class, the parameter FormObj is received from the *new* expression in the form constructor, and simply passed on to the Select class, which in turn handles the form binding. When the Form Designer creates the custom component class, it uses the parameter name FormObj; but the parameter could have any name. The important thing is that the parameter received by the subclass is passed on to the base class.

Custom component properties and methods

Related topics

Most of the properties saved by the Form Designer are not needed for the custom class. The position properties, including the *pageno*, are unnecessary, Because they will be set when the component is used. The *dataLink* is also not needed, because that too will be set, if needed, in each form.

Setting the *onServerLoad* property is also redundant. In the Form Designer it was necessary to create a method with the component name and the event, and assign that method to the component's event property—because the method created is actually a form method; it appears in the form's *class* definition.

On the other hand, because we are now creating a *class* definition for the component itself, all you need to do is change the name of the method to *onServerLoad*, and it will automatically be used as the object's *onServerLoad* event handler.

Adding custom properties to custom components

Related topics

In order for the FieldSelect component to operate generically, it needs to know which field in which rowset to read. These two items are examples of custom properties; in other words, properties that are not part of the base class.

Initialize these properties in the component's constructor. You cannot use a *with* block to create properties—it is used to refer or assign values to existing properties only—so create them using the *this* reference and dot notation. Then set them before the component's *onServerLoad* event is fired, so that the event handler will read the right field. If they are not initialized in the component's constructor, you cannot easily detect that they have not been set, which can lead to run-time errors.

Custom component definition

Related topics

The definition for the FieldSelect class, including modifications to the *onServerLoad* event handler to use the custom rowset and field properties should be changed to:

```
class FieldSelect(FormObj) extends Select(FormObj) custom {
   this.rowset = null;  // Create new custom properties
   this.field  = "";
   function onServerLoad()
   {
     if ( this.rowset != null ) {
       this.aOptions = new Array();
       this.rowset.first();
       while ( !this.rowset.endOfSet ) {
         this.aOptions.add( this.rowset.fields[ this.field ].value );
         this.rowset.next();
       }
       this.options = "array this.aOptions";
   }
}
```

Make the changes to SELECT.CC, save the file, and close the Script Editor.

Using custom components

Related topics

To use custom components, the CC file that contains the class definition must be loaded into memory. If you use the Setup Custom Components dialog box as detailed in Custom components, the listed files will be loaded into memory when you start IntraBuilder.

The act of opening a script file in the Script Editor automatically unloads it from memory—even if no changes are made. In that case, you can always load the script manually in the Script Pad by calling the *_sys.scripts.load*() method; for example:

```
_sys.scripts.load( "SELECT.CC" )
```

In the Form Designer, all the custom components loaded into memory will appear on the Custom tab of the Component Palette.

When a custom component is added to a form in the Form Designer, a *_sys.scripts.load*() method call is added at the top of the form constructor, so that the CC file that contains the custom component will always be loaded when the form is run or edited in the Form Designer.

Replacing an existing component with a custom component

Related topics

To replace an existing component with a custom component, you can always simply delete the old component and drag a new custom component from the Component Palette in the Form Designer.

Another technique is to change the class name in the *new* expression that creates the component. This is useful because it preserves all the existing properties, such as its position and *dataLink*, and is the only way to switch a form component from one class to another.

To use the new FieldSelect component in the Viewer form,

1. In the Forms page of the IntraBuilder Explorer, right-click VIEWER.JFM and choose Edit as Script from the context menu.
2. Use the Script Editor's Find feature to locate the *new* expression that creates the sectionSelect component (search for "sectionSelect").

```
with (this.sectionSelect = new Select(this)){
```

3. Change the class name from Select to FieldSelect.

```
with (this.sectionSelect = new FieldSelect(this)){
```

4. Remove the component's *onServerLoad* property assignment, so that the component will use the *onServerLoad* handler defined in the SELECT.CC file, not the one defined in the VIEWER.JFM file.
5. The *with* statement that creates the component should now look like this:

```
with (this.sectionSelect= new FieldSelect(this)){
   left = 30;
   top = 3;
   width = 20;
   dataLink = parent.query1.rowset.fields["Section"];
   pageno = 0;
}
```

6. Find the sectionSelect_onServerLoad() method and delete it; it's no longer needed. Be sure not to delete any adjacent methods.
7. When adding a new message, the first section is used as the default. The newButton_onServerClick() method gets the first elements of the aSections array of the sectionSelect component. Since the FieldSelect class is now used instead, and its array is named aOptions, the array name in the method must be changed. Make the highlighted change to the end of the second statement in the method:

```
function newButton_onServerClick()
{
 this.form.rowset.beginAppend();
 this.form.rowset.fields[ "Section" ].value =
this.form.sectionSelect.aOptions[ 0 ];
 this.form.compose( "New" );
}
```

8. Because you're adding the custom component manually, you'll also need to load the SELECT.CC file. Go to the top of the *class* definition. Between the *class* line and the first *with* statement, insert the highlighted line:

```
class viewerForm extends tmdbaseCForm from "TMDBASE.JCF" {
   _sys.scripts.load( "SELECT.CC" );
   with (this) {
```

Setting custom properties

Related topics

Finally, for the FieldSelect component to actually work, you need to set the component's rowset and field properties. Because these properties are initialized in the component's constructor, they appear in the Inspector (under JavaScript Variable Properties in the Inspector outline).

You can set the component's custom properties in the form's *onServerLoad* event. The form's *onServerLoad* fires before the *onServerLoad* for any components.

In the Script Editor, add the highlighted lines to the end of the Form_onServerLoad() method:

```
function Form_onServerLoad()
{
  try {
    // Get user ID and name from login form
    this.userRowset= VIEWER.arguments[ 0 ].rowset;
    this.userQuery = this.userRowset.parent;
    this.userID     = this.userRowset.fields[ "User ID" ].value;
    this.userName   = this.userRowset.fields[ "User name" ].value;
    this.HMN        = this.userRowset.fields[ "HMN" ].value;
  }
  catch ( Exception e ) {
    this.userID   = 0 ;
    this.userName = "Unregistered" ;
    this.HMN      = 0;
  }
  // Set "From" name on compose page to user name, because it will never
change
  this.userLabel.text = this.userName;
  this.applyHMN();
  this.refreshUnlinked();
  this.sectionSelect.rowset = this.sections1.rowset;
  this.sectionSelect.field  = "Name";
}
```

Custom data access components

Related topics

In addition to custom visual components that you can reuse on forms, you can also subclass data access components.

For example, you can create a subclass of the Query class that contains the *sql* property to access a table and all the event handlers needed to implement field morphing for that table. This encapsulates all the attributes of the table, allowing you to treat the table simply as a reusable object.

The TMD project: Reusing the Messages table

Related topics

The data access objects that represent the Messages table in the Viewer form already use a number of event handlers to implement field morphing for the Section and From fields. The field morphing will be useful in other forms and reports, so we'll save the Messages table in the Viewer form as a custom component:

1. Open VIEWER.JFM in the Form Designer.
2. Click the messages1 object.
3. Choose File|Save as Custom... from the menu.
4. In the Save as Custom dialog box, make sure that Save Select Components as Custom is selected.
5. Set the Class Name to "messagesQuery".
6. Set the Custom Component File to "TMD.CC".
7. Click OK.
8. Close the Form Designer.

Examining the custom data access component definition

Related topics

In the Custom tab of the IntraBuilder Explorer, right-click TMD.CC. Choose Edit as Script from the context menu. The JavaScript code should look like this:

```
class messagesQuery extends Query custom {
   with (this) {
      left = 69;
      top = 0;
      sql = 'SELECT * FROM "messages.db"';
      active = true;
   }
   with (this.rowset) {
      fields["From"].beforeGetValue = class::messages1_from_beforeGetValue
      fields["Section"].canChange = class::messages1_section_canChange
      fields["Section"].beforeGetValue =
class::messages1_section_beforeGetValue
      indexName = "Thread";
      canAppend = class::messages1_canAppend;
      canSave = class::messages1_canSave;
      onAbandon = class::browse;
      onAppend = class::messages1_onAppend;
      onEdit = class::messages1_onEdit;
      onNavigate = class::messages1_onNavigate;
      onSave = class::browse;
   }
   function messages1_onEdit()
   {
     this.parent.parent.pageno = 2;// Display the editing page
   }
   function browse ()
   {
     this.parent.parent.statusLabel.visible = false;
     this.parent.parent.pageno = 1;// Display the browsing page
   }
   function messages1_canSave()
   {
     var lRet = true;// Logical return value defaults to true; assume
everything is OK
     var cErrors = ""; // Text to contain errors
     if ( this.fields[ "From" ].value == null ) {
       lRet = false;
       cErrors += "- <B>From</B> field cannot be blank<BR>";
     }
     if ( this.fields[ "To" ].value == null ) {
       lRet = false;
       cErrors += "- <B>To</B> field cannot be blank<BR>";
     }
     if ( this.fields[ "Subject" ].value == null ) {
       lRet = false;
       cErrors += "- <B>Subject</B> field cannot be blank<BR>";
     }
     if ( !lRet ) {  // If there are problems set the error message
       this.parent.parent.statusLabel.text = "The data cannot be saved
because<BR>" +
```

```
                                        cErrors;
  }
  this.parent.parent.statusLabel.visible = !lRet;  // and display it
  return lRet;    // Return success or failure
}
function messages1_onAppend ()
{
  this.fields[ "From" ].value = this.parent.parent.userID;
  this.fields[ "# of replies" ].value = 0;
  this.modified = false;
  this.parent.parent.pageno = 2;// Display the editing page
}
function messages1_onNavigate()
{
  this.parent.parent.refreshUnlinked();
  this.parent.parent.updateHMN();
}
function messages1_from_beforeGetValue()
{
  if ( this.parent.parent.endOfSet ) {
    // When navigating to end-of-set
    return null;
  }
  else if ( this.value == null ) {
    // For beginAppend()
    return "";
  }
  else {
    // Normal lookup, with value in case lookup fails
    var r = this.parent.parent.parent.parent.users1.rowset;
    return r.applyLocate( '"User ID" = ' + parseInt( this.value ) ) ?
           r.fields[ "User name" ].value : "Unregistered";
  }
}
function messages1_canAppend()
{
  this.parent.parent.bookmark = this.bookmark();
  return true;
}
function messages1_section_beforeGetValue()
{
  if ( this.parent.parent.endOfSet ) {
    // When navigating to end-of-set
    return null;
  }
  else if ( this.value == null ) {
    // For beginAppend()
    return "";
  }
  else {
    // Normal lookup, with value in case lookup fails
    var r = this.parent.parent.parent.parent.sections1.rowset;
    return r.applyLocate( '"Section #" = ' + parseInt( this.value ) ) ?
           r.fields[ "Name" ].value : "Closed section";
  }
}
```

```
    function messages1_section_canChange( newValue )
    {
      var r = this.parent.parent.parent.parent.sections1.rowset;
      if ( r.applyLocate( '"Name" = \'' + newValue +'\'' ) ) {
        this.value = r.fields[ "Section #" ].value;
      }
      return false;
    }
}
```

Like the custom visual component, the custom data access component class contains the property settings and methods, not just for the Query object itself, but also for its rowset and fields.

As you can see, there is a lot of code that can be removed from the VIEWER.JFM file, because the methods are now defined in the TMD.CC file. But that doesn't include all the code that was streamed out. The *beforeGetValue* and *canChange* methods used for field morphing should be kept in the custom class, because they directly affect the representation of the table. But the rowset events, such as *canAppend*, *onAppend*, and *onNavigate* events are not needed; they're more involved with the form.

You can use the custom data access component in the form with its preset field morphing event handlers, and add the event handlers for the rowset.

Making the custom data access component generic

Related topics

As saved by the Form Designer, the custom class relies on some hard-coded references to other Query objects for the User name and Section lookups. For the messageQuery class to be truly reusable, these hard-coded references must be replaced by references that can be set at runtime.

In addition, we need to clean up the code in several places. The steps listed here are specifically for the messageQuery class above, but demonstrate the typical changes needed to make a custom data access component generic:

1. The position properties are not needed in the definition of the query. You set the position when the query is used in the Form Designer, not in the custom class definition. This leaves only the *sql* and *active* property assignments in the *with*(*this*) statement. You must activate the query in the custom class so that event handlers can be assigned to its fields. If the query is not activated, the rowset is empty and has no fields. The rest of the properties should be removed.
2. For mainly cosmetic reasons, you can remove the name of the query from the method names for fields' *beforeGetValue* and *canChange* event handlers. This must be done both in the assignment of the event handlers in the *with*(*this.rowset*) statement and the *function* definitions.
3. The rowset event handlers are not needed, so you should remove both the assignment of the event handlers in the *with*(*this.rowset*) statement and the *function* definitions.

Note The *onEdit* event handler isn't used by the TMD Viewer form either; it's left over from Using common event handlers. You can safely remove both the event handler assignment and the *function* definition from the Viewer form if you want. It's simply unused code.

4. Instead of the hard-coded references to the User name and Section queries, the messageQuery object needs two custom properties, which you can set in its *onOpen* event.

The cleaned-up custom data access object class with the new *onOpen* event handler looks like this:

```
class messagesQuery extends Query custom {
   with (this) {
      sql = 'SELECT * FROM "messages.db"';
      active = true;
   }
   with (this.rowset) {
      fields["From"].beforeGetValue = class::from_beforeGetValue
      fields["Section"].canChange = class::section_canChange
      fields["Section"].beforeGetValue = class::section_beforeGetValue
      indexName = "Thread";
   }
   function onOpen()
   {
     with ( this.qUsers = new Query() ) {
       sql = "select * from USERS.DB";
       active = true;
     }
     with ( this.qSections = new Query() ) {
       sql = "select * from SECTIONS.DB";
       active = true;
     }
   }
   function from_beforeGetValue()
   {
     if ( this.parent.parent.endOfSet ) {
       // When navigating to end-of-set
       return null;
     }
```

```
    else if ( this.value == null ) {
      // For beginAppend()
      return "";
    }
    else {
      // Normal lookup, with value in case lookup fails
      var r = this.parent.parent.parent.qUsers.rowset;
      return r.applyLocate( '"User ID" = ' + parseInt( this.value ) ) ?
             r.fields[ "User name" ].value : "Unregistered";
    }
  }
  function section_beforeGetValue()
  {
    if ( this.parent.parent.endOfSet ) {
      // When navigating to end-of-set
      return null;
    }
    else if ( this.value == null ) {
      // For beginAppend()
      return "";
    }
    else {
      // Normal lookup, with value in case lookup fails
      var r = this.parent.parent.parent.qSections.rowset;
      return r.applyLocate( '"Section #" = ' + parseInt( this.value ) ) ?
             r.fields[ "Name" ].value : "Closed section";
    }
  }
  function section_canChange( newValue )
  {
    var r = this.parent.parent.parent.qSections.rowset;
    if ( r.applyLocate( '"Name" = \'' + newValue +'\'' ) ) {
      this.value = r.fields[ "Section #" ].value;
    }
    return false;
  }
}
```

In the *onOpen* event, the two queries for the user name and section name lookup are created as custom properties of the custom query object. Because they are part of the query object, they're unrelated to any Query objects that might open the same tables on the form. It also means you don't have to open these tables yourself unless you're doing some editing on those tables.

In the *beforeGetValue* and *canChange* event handlers, the embedded queries are used for the lookups. The references which are assigned to the *var* r have one less *parent*, since you only have to go to the query, not the form, to access them.

Integrating reports introduction

Related topics

This series of topics examines report sorting and explains how to integrate forms created in the Report Designer into your web application. For an introduction to the Report Designer, see Report Designer

Report sorting and grouping

Related topics

Because reports are intended to display data in a useful and informative format, the order in which data is sorted in a report is usually a design consideration.

It is especially important if the report contains groups. All the rows for the same group must be contiguous. For example, when breaking down sales by state, all the rows for each state must be together.

Group objects

Related topics

Report grouping is implemented with Group objects. These objects are contained in the report's StreamSource object. Nested groups are handled in the order that they are created.

Each Group object has a *groupBy* property. This property contains the name of a field. This field can be:

- An actual field in the table
- A calculated SQL field created in the SQL SELECT statement, or
- A CalcField object

As IntraBuilder's report engine traverses the rowset, it watches the *value* of the specified fields. Every time the *value* changes, another group starts.

Therefore, all rows in a group must be contiguous. The rows in the rowset must be sorted in the correct order.

Controlling the sort order

Related topics

There are two ways to control sort order:

- With an ORDER BY clause in the query's SQL SELECT statement.
- With the query's rowset's *indexName* property, if the table in the query is a Standard (DBF or DB) table.

You can use ORDER BY for any combination of fields. For example, if you are grouping by state and then zip code, you should specify those fields in the SQL SELECT statement in the query's *sql* property:

```
select * from SALES order by STATE, ZIP
```

The *indexName* property can be used only for queries based on DBF or DB tables.

The *indexName* property can be set to any existing index tag. For example, if the index tag is on the state and zip code fields, and the name of that tag is STATE_ZIP, you would set the rowset's *indexName* property to "STATE_ZIP".

autoSort property

Related topics

If a Report object's *autoSort* property is true (the default), then the query's *sql* property will be modified automatically to include an ORDER BY clause that sorts the rowset in the correct order.

For example, if you have two Group objects, the first grouping by the field State and the second by Zip, then even if the query's *sql* property is set as

```
select * from SALES
```

the rowset will actually be generated internally with the SQL statement

```
select * from SALES order by STATE, ZIP
```

If *autoSort* is false, the rowset is not altered by the report engine. It assumes that the query is correct and contains the necessary fields in the right order. Therefore, if you use the *indexName* property to set the rowset order, you should set *autoSort* to false; otherwise it defeats the purpose of using *indexName*.

Grouping with morphed fields

Related topics

The sort order and the value of the group fields are actually independent of each other. In most cases, they happen to coincide—the rowset is sorted on those fields, so field values appear sorted.

But if you are sorting on morphed fields, then the data is sorted on the raw value of the field in the table, while the apparent value displayed in the report would be the morphed value.

For example, if you're sorting messages by section, and the sections are:

Section #	Name
1	General
2	Dogs
3	Cats

and Section is a morphed field which stores the section # but displays the name, as detailed in One-way field morphing, then the sections will be sorted in numerical order, because the numbers are in the table, and the sections will appear to be out-of-order. This is often the desired effect.

If you want the sections to appear sorted on the morphed value of the field—in this example, Cat, Dog, and General—you'll need to sort on that field. You can't use morphed field values in an index or SQL ORDER BY, so you'll need to use a SQL JOIN, like:

```
select M.*, S.name as SectionName from MESSAGES.DB as M join SECTIONS.DB as S
  on M.section = S."section #" order by SectionName
```

This SQL SELECT statement joins the MESSAGES.DB and SECTIONS.DB table, aliased as M and S respectively. They are joined between M's Section field and S's Section # field. All the fields from M are selected, and S's Name field is aliased as the field name SectionName. The result is sorted by the SectionName field.

The TMD project: A report of new messages

Related topics

We would like the Threaded Message Database to give the users a list of new messages when they log in.

Use the Report Expert to quickly layout a report of new messages and go into the Report Designer to polish it. The result should look like the figure at the bottom of this topicFigure 15.1.

1. In the Reports tab of the IntraBuilder Explorer, double-click the (Untitled) report.
2. Choose the Report Expert.
3. Select MESSAGES.DB from the file list and click Next.
4. Make sure Include Detail Rows is selected and click Next.
5. Select the Message #, From, To, Subject, and Posted fields and click Next.
6. Group on the Section field. Make sure it's ascending and click Next.
7. Create a Count of Message # summary for both the Grand summary and the Section group and click Next.
8. Change the report title to "New messages". Make sure the report is Tabular and Continuous. Include the Date, but remove the Page number and click Next.
9. Click Design Report.
10. Select the streamSource1.group1.headerBand object. If it is not visible, use the drop-down selection list in the Inspector.
11. Set its height property to 250.
12. Drag a HTML component from the component palette and drop it in the streamSource1.group1.headerBand.
13. Set its *text* property type to CodeBlock and its value to:

```
  {||"Section: " +
this.parent.parent.parent.rowset.fields[ "Section" ].value}
```

14. Select that HTML component, the report title, and date, and set their width property to 9360 so that they span the width of the page.
15. Modify the string at the beginning of the *text* property of the streamSource1.group1.footerBand.HTML1 component so that the property reads (all in one line):

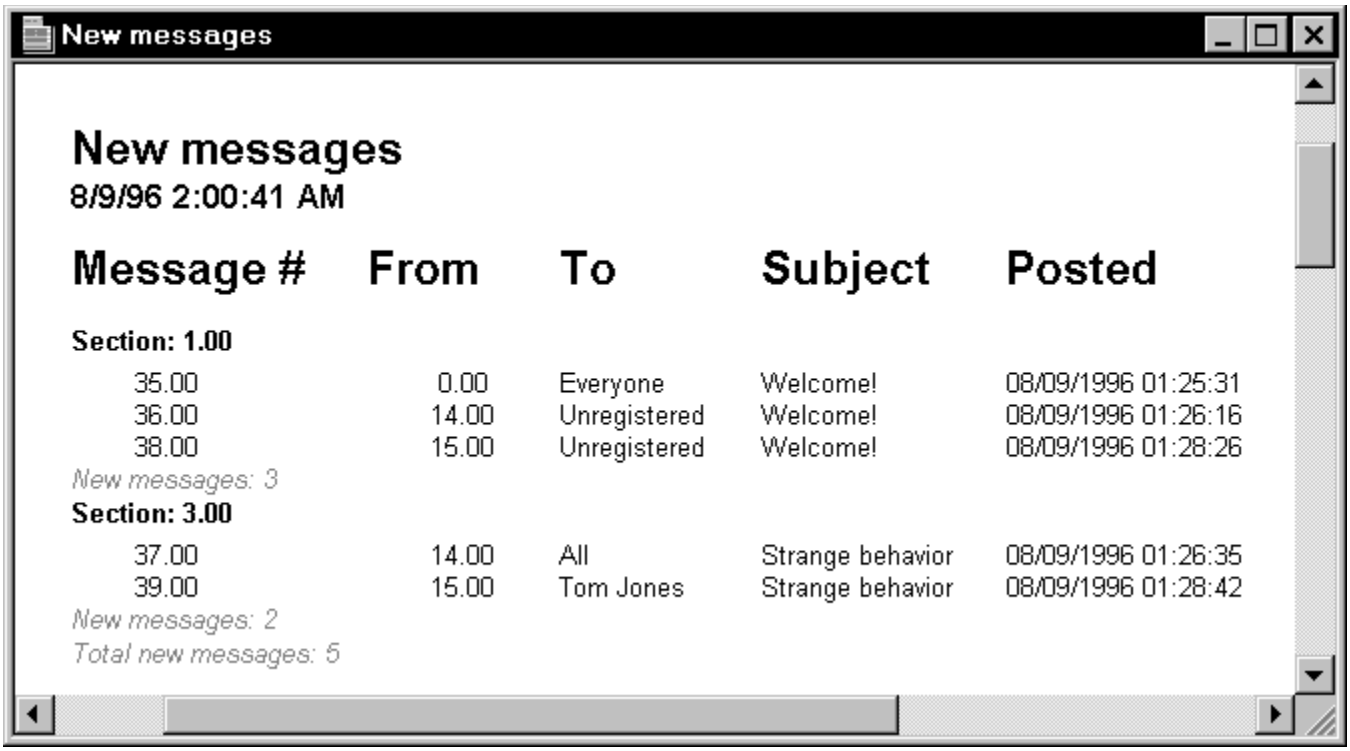
```
  {||"New messages: " + this.parent.parent.agCount({||
this.parent.rowset.fields["Message
  #"].value})}
```

16. Modify the string at the beginning of the *text* property of the streamSource1.group1.footerBand.HTML1 component so that the property reads (all in one line):

```
  {||"Total new messages: " + this.parent.parent.agCount({||
  this.parent.streamSource1.rowset.fields["Message #"].value})}
```

17. If necessary, move the Subject and Posted columns to the right and widen the To column so that the names don't wrap.
18. Click the run button in the toolbar. When prompted for a file name, use "newmsg" (the JRP extension will be added automatically).

Notice that the From and Section fields display as their numbers, not their names. This is because the report's query uses the plain MESSAGES.DB table, without all the field morphing that was setup in Two-way field morphing. Close the report preview window before proceeding.



The New Messages report

Using custom data access components in reports

Related topics

In The TMD project: Reusing the Messages table, the complete Messages query was saved as a custom data access component for easy reuse. Now is the time to reuse this custom component.

The Report Designer does not support custom components in the same way as the Form Designer. Specifically, you cannot save report components as custom components, nor the entire report as a custom report, nor can you access the custom component setup in the Report Designer.

If custom components are loaded into memory, however, they will appear in the Custom tab of the Component Palette and can be dropped onto reports, provided the root class is one of those that can be placed on a report.

For example, you cannot place Button controls on a report, so if you have any custom buttons, you can't put those on reports either. On the other hand, you can use HTML and Query objects on reports, so you can reuse any custom HTML or Query objects that you created from the Form Designer or from scratch.

Replacing the existing data access component

Related topics

Because there's already a query in the report, the easiest way to include the custom messagesQuery object is to edit the JRP file:

1. Right-click the NEWMSG.JRP file in the Reports tab of the IntraBuilder Explorer and choose Edit as Script from the context menu.
2. Use the Script Editor's Find feature to locate the *new* expression that creates the messages1 component:

```
with (this.messages1 = new Query()){
```

3. Change the class name from Query to messagesQuery:

```
with (this.messages1 = new messagesQuery(this)){
```

4. Remove the component's *sql* property assignment, because that's already defined in the custom class.
5. The *with* statement that creates the component should now look like this:

```
with (this.messages1 = new messagesQuery()){
   left = 0;
   top = 0;
   active = true;
}
```

6. Because you're manually adding the custom component, you'll also need to load the TMD.CC file. Go to the top of the *class* definition. Between the *class* line and the first *with* statement, insert the highlighted line:

```
class NEWMSGReport extends Report {
   _sys.scripts.load( "TMD.CC" );
   with (this) {
```

7. The messageQuery object is already sorted in Thread order, so you don't want the report engine overriding that. Set the report's *autoSort* property to false at the beginning of the *with* (*this*) block:

```
class NEWMSGReport extends Report {
   _sys.scripts.load( "TMD.CC" );
   with (this) {
      autoSort = false;
```

8. By using the messagesQuery object, the From field is displayed as the User name instead the number, so the *alignHorizontal* property of the field must be changed from Right (as generated by the Report Expert) to Left. You can make this change with the Inspector in the Report Designer, but while you have the JRP file open, you can make the change in the script file. Find the *with* block that defines streamSource1.detailBand.HTML2 (use the editor's Find dialog and look for "detailBand.HTML2"):

```
   with (this.streamSource1.detailBand.HTML2 = new
HTML(this.streamSource1.detailBand)){
      Ä
      alignHorizontal = 2;
      Ä
   }
```

   Find the statement that assigns the *alignHorizontal* property (2 is the enumerated value for Right) and delete it. This causes the horizontal alignment to be the default, which is Left.
9. Save the file and close the Script Editor.

Now run the report.

Just by using the custom query component, the From and Subject fields display the names instead of the numbers, without having to modify the field values in the report or bother with any lookup tables.

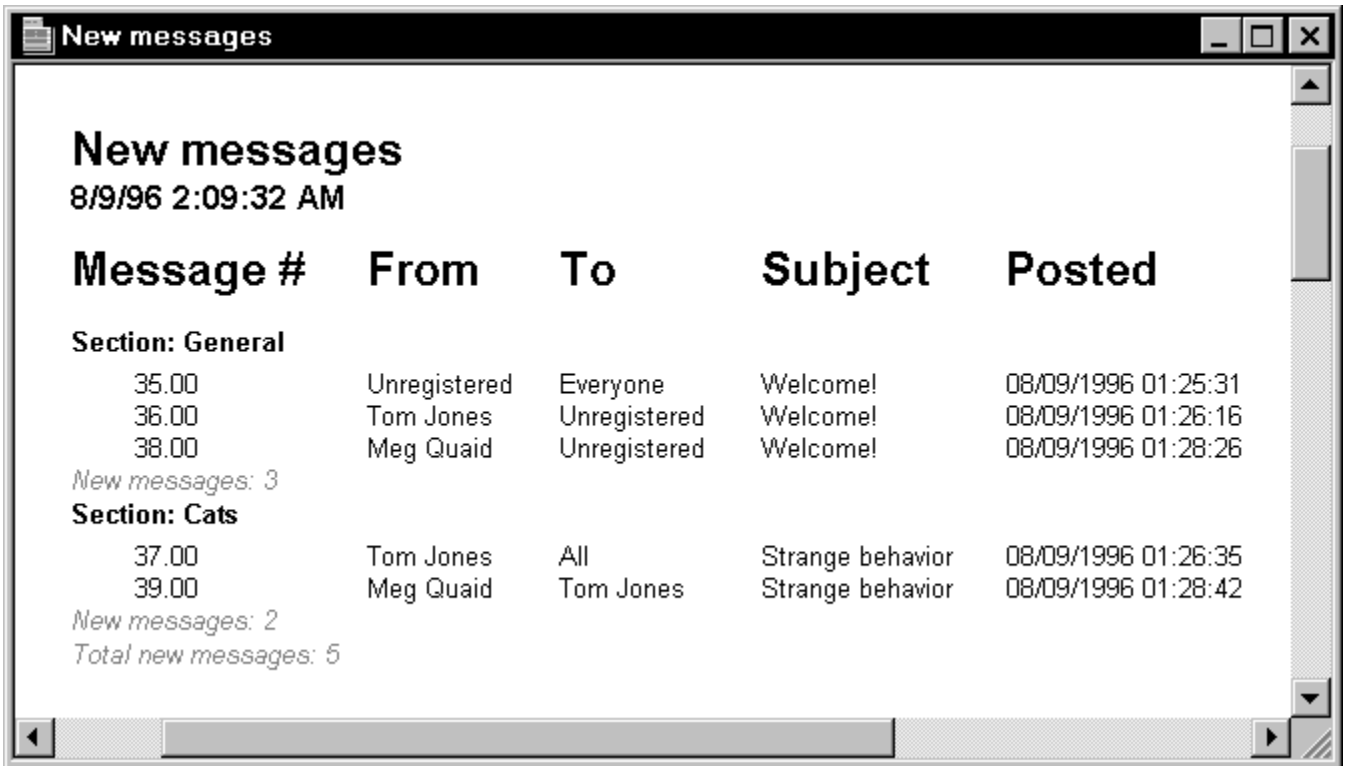
If you had used the Designer to create the report from scratch, it would have been easier to drop the custom Query component from the palette onto the report. But in a case like this, using the Report Expert to lay out the fields saves you more effort than would be required to insert the custom Query object manually.

Calling reports from forms

Related topics

In many ways, reports are similar to forms.

For example, to call a report from a form's event handler in a Button control's *onServerClick*, call the *_sys.reports.run*() method, which is almost identical to the *_sys.forms.run*() method. The only difference is that the *reports* method assumes a JRP extension, while the *forms* method assumes a JFM extension.



New messages report with custom query object

The TMD project: Calling the New Messages report

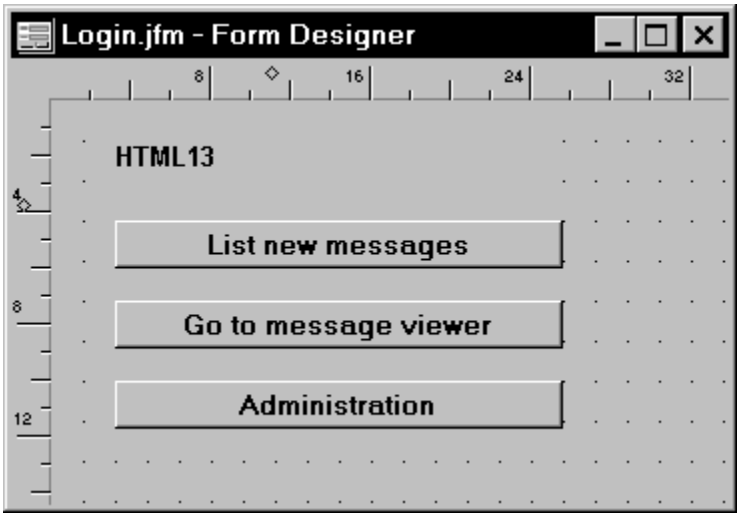
Related topics

When the user logs in, if there are new messages, the user is given the option of displaying a list. There are already separate pages in the Login form that are displayed depending on whether or not there are new messages when the user logs in. The “There are new messages” page must be augmented to support the report.

1. Open LOGIN.JFM in the Form Designer.
2. Go to page 4, which contains the button that starts the Viewer form.
3. Add a Button control above the Viewer button. Set its name to listNewButton and its *text* to “List new messages”.
4. Set listNewButton’s *onServerClick* event to:

```
function listNewButton_onServerClick()
{
  _sys.reports.run( "NEWMSG", this.form );
}
```

Switch the form back to page 1, save the form, and close the Form Designer.



The "There are new messages" page of the Login form with the List new messages button

Running code when opening a report

Related topics

When listNewButton is clicked, NEWMSG.JRP is called via the *_sys.reports.run*() method. Like a JFM form file, a JRP report file contains bootstrap code that instantiates and runs the report. The bootstrap code for a report does a bit more, though.

Default report parameters

Related topics

Open NEWMSG.JRP in the Script Editor and examine the bootstrap code between the header and the *class* definition:

```
// {End Header} Do not remove this comment//
// Generated on 08/05/96
//
var r = new NEWMSGReport();
if (NEWMSG.arguments.length == 2) {
   r.startPage = NEWMSG.arguments[0];
   r.endPage = NEWMSG.arguments[1];
}
r.render();
class NEWMSGReport extends Report {
```

You can see that between the instantiation of the NEWMSGReport object and calling its *render*() method, the bootstrap code looks for two arguments passed to the NEWMSG.JRP file in the script's *arguments* array.

By default, you can pass a starting and ending page in the call to a report. The first parameter is assigned to the report's *startPage* property, and the second parameter is assigned to its *endPage* property.

When listNewButton calls the report, it passes a single parameter, a reference to the form, just like when calling the Viewer form. So the default bootstrap code cannot be used.

Custom header code

Related topics

When the Login form calls the Viewer form, the Viewer form uses its *onServerLoad* event to get the form reference passed by the Login form. Reports do not have *onServerLoad* events, but they do have a *preRender* event that is similar, in that it is an event that runs once before it is displayed.

Because the standard report bootstrap code looks for exactly two parameters, the *startPage* and *endPage* properties would not be affected when calling the report with the single form reference parameter. On the other hand, if at some later date another parameter were added, the standard bootstrap code would definitely get in the way.

Therefore, a better solution would be to use your own bootstrap code instead of the default. Remember that it's not worth changing the default bootstrap code because it gets rewritten every time the report or form is saved. What you can do is put your own code in the header.

The TMD project: Incorporating the User ID in the report

Related topics

Type in the following above the // {End Header} line:

```
var r = new NEWMSGReport();
try {
  var f = NEWMSG.arguments[ 0 ].rowset.fields;
  r.userID   = f[ "User ID"   ].value;
  r.userName = f[ "User name" ].value.toUpperCase();
  r.HMN      = f[ "HMN"       ].value;
}
catch ( Exception e ) {
  r.userID   = 0;
  r.userName = "Unregistered" ;
  r.HMN      = 0;
}
r.streamSource1.rowset.filter = '"Message #" > ' + parseInt( r.HMN );
r.render();
return;
// {End Header} Do not remove this comment//
```

As you can see, all the code should be placed above the {End Header} comment. Equally important is the fact that the custom header code ends with a *return* statement. If the *return* were not there, the default bootstrap code would continue to run after the custom code in the header, resulting in two separate reports.

The custom header code is a variation of the *onServerLoad* code used by the Viewer form. It works like this:

1. The report object is instantiated, and a reference is stored in the variable r, just like the standard bootstrap.
2. The custom header code uses a *try* block to catch errors. The most likely error is that the report was run directly rather than through the Login form.
3. The custom header code assigns a reference to the *fields* array for the user's row in the Users table to the variable f.
4. The custom header code assigns the user's ID, name, and HMN (High Message Number) as properties of the report. The user name is stored in uppercase to expedite name matching later.
5. The custom header code sets the rowset's *filter* property to display only the new messages, using the same filter condition used in the Login form's checkNewMessages() method.
6. Finally, the report is rendered.

Reports and the form stack

Related topics

There is one small problem: when a report is rendered, its contents are displayed in the client browser, but it doesn't go on the form stack.

In fact, there are no direct links to server-side events once the report has been rendered. For example, reports cannot have buttons, and although you can have images on reports, their *onImageServerClick* events do not fire for reports.

This means that although the report has taken over the browser window, there is no direct way to remove the report in the same way you could close a form and see the form underneath it in the form stack.

The user can still use the browser's Back button to go back to the previous form. You can also code HTML links in the report. These links can contain URLs for other IntraBuilder forms and reports.

Finally, you can use client-side JavaScript to make your reports more interactive. This is demonstrated in Client-side JavaScript.

Conditionally highlighting rows in a report

Related topics

Now that the report knows who is requesting it, any messages addressed to the user can be highlighted.

Each visual component on the form has a *canRender* event that is fired before the component is rendered. You can use this event to prevent the rendering of a component by returning false, but more useful is the ability to change the properties of the component before it is rendered. In that case, the event handler should return true.

The TMD project: Highlighting a user's waiting messages

Related topics

Follow these steps to add highlighting to the report:

1. Open NEWMSG.JRP in the Report Designer.
2. Create a new method:

```
  function highlightWaiting()
  {
    this.color =
this.form.messages1.rowset.fields[ "To" ].value.toUpperCase() ==
                this.form.userName ? "red" : "black";
    return true;
  }
```

3. Link this new method to the *canRender* event of the From, To, Subject, and Posted fields (the HTML objects HTML2 through HTML5 in streamSource1.detailBand). The Message # (HTML1) doesn't need to be linked.

Every time the field is rendered, the highlightWaiting() method is executed. It compares the value of the To field in the current row with the value stored as a property of the report in the custom report header. Note that when a visual component is placed in a report, its *form* property refers to the report.

If they match, the color is set to red to indicate a waiting message for the user. If they don't match, the color is set to black, the normal color.

The name of the user that's running the report was stored in uppercase in the header, so the field name is converted to uppercase as well for the match. This means that users don't have to get the capitalization of the user names exactly correct for the TMD to consider that a waiting message.

If you run the report directly from the IntraBuilder Designer without going through the Login form, you will be considered an unregistered user. This is handy for testing. Since the user name is stored in proper case as "Unregistered", the report will never find waiting messages for unregistered users.

Client-side JavaScript

Related topics

To reach the broadest number of users, your IntraBuilder applications may rely solely on server-side JavaScript. Server-side JavaScript enables you to build complete Web-based database applications.

On the other hand, client-side JavaScript can be used to build additional functionality, create advanced applications, and partition your application logic.

A complete discussion of client-side JavaScript is beyond the scope of this guide. Entire books have been written to describe how to use JavaScript to enhance Web pages. This series of topics focuses on how IntraBuilder supports client-side JavaScript, how to interface it with server-side JavaScript, and how it can enhance your applications.

A brief client-side JavaScript primer

Related topics

Client browsers support JavaScript through the use of HTML tags. All scripts are embedded between two tags:

```
<SCRIPT>
// JavaScript code here
</SCRIPT>
```

The text between the <SCRIPT> tags is considered to be JavaScript. To be more accurate, the <SCRIPT> tag is intended to support all present and future HTML scripting languages, so the language name should be declared.

Browsers that do not support JavaScript will ignore the <SCRIPT> tags and display the script text as plain text. For this reason, it is a good idea to embed JavaScript inside an HTML comment:

```
<SCRIPT LANGUAGE="JavaScript">
<!-- hide code from non-JavaScript browsers
// JavaScript code here
// The next line is both a JavaScript comment and the end of an HTML comment
//-->
</SCRIPT>
```

JavaScript code can be placed anywhere in an HTML document. It is executed as it is encountered in the rendering of the document.

Here is an important distinction between client-side and server-side JavaScript: With client-side JavaScript, functions are not available until their code is encountered. In other words, when a function is encountered, it is interpreted and stored and is available for execution. A call to a function may not precede the function definition in an HTML document, whereas it could in a server-side script.

Therefore, function definitions are almost always placed in the <HEAD> section of an HTML document, so that they are read first, before any of the <BODY> of the document.

Note The server-side JavaScript extensions available in IntraBuilder are not available when running code server-side. In particular, classes and exception handling are not available.

Exporting JavaScript methods

Related topics

IntraBuilder supports client-side JavaScript through the use of exported methods.

Methods written in form and report classes can be exported so that they appear in the <HEAD> of the dynamically formulated HTML document. All exported methods appear together in a single <SCRIPT> block.

To export a method, the following line must be the first line in the method:

```
// {Export} This comment causes this function body to be sent to the client
```

The Method Editor in the Form and Report Designers automatically adds this line for any new methods it creates. You can erase this line if you intend the method for server-side execution and you don't want your code exported to the client.

IntraBuilder also automatically exports any methods tied to client-side events, such as a Button control's *onClick* or TextArea control's *onBlur*. These event handlers are executed on the client browser when running the form from the client, but they will also run on the server when testing the form locally.

Because server-side JavaScript is a superset of client-side JavaScript, all client-side JavaScript written in form and report classes should compile without errors.

While the client-side JavaScript is syntactically correct, IntraBuilder lacks a number of browser elements such as a history object and frames, which are programmable through client-side JavaScript. Therefore, some client-side code will cause run-time errors. In most cases, these errors can be ignored, although the action of these statements obviously will not occur.

This means that the only truly effective way to test client-side code is through a client browser.

On the other hand, you should make sure you do not export methods that contain server-side JavaScript extensions, such as *try* blocks. This would cause your client-side code to fail, because the browser cannot interpret these extensions.

The this reference

Related topics

JavaScript is a fairly loose and free-form language.

Because client-side JavaScript does not support formal classes in the same way IntraBuilder, you will often see the use of *function*s as general functions instead of methods and the passing of the *this* reference as a parameter to these functions. While certainly legal, this practice sacrifices object-orientation. You cannot be sure what the *this* reference means inside these functions, because they are no longer bound to the objects that called them.

When IntraBuilder exports a method, it guarantees that the *this* reference inside the function refers to the object which called the method. In other words, you can write object-oriented code for client-side JavaScript in the same way you do for server-side JavaScript in IntraBuilder.

Client-side validation

Related topics

As discussed in Validating data,, data validation is the most important aspect of a database application. You should always validate data before it is saved to your tables.

When using server-side JavaScript on IntraBuilder to validate data, the user does not know if there are mistakes or missing items in the data entry until they submit the form.

Using client-side JavaScript to do client-side validation has two benefits:

- The user gets immediate feedback on their actions, on a field-by-field basis if desired, without having to wait for the form to be submitted and returned.
- Putting validation code on the client is a form of application partitioning. This divides the processing between the various tiers of your application. It offsets some of the work load that the IntraBuilder application server would have to do, enabling it to more quickly handle server requests.

Even if you expect to use client-side validation, it's possible that the user is accessing your Web site with a browser that does not support JavaScript, or the user has disabled JavaScript in their browser. Because data integrity is so critical, you should always validate the data with server-side code, and use client-side validation to augment the server-side checks. If the client-side code is working, the server will never have to deal with an error; but if the client is not validating data, the server will catch any problems.

Client-side validation events

Related topics

Client-side validation centers around the following events:

- *onChange*
- *onBlur*
- *onSubmit*

onChange event

Related topics

The *onChange* event fires when a Select, Text, or TextArea component loses focus, provided that the contents of the component have been changed.

This is not suitable for checking whether the field is blank, because the most likely reason that a field is blank is because it was left blank; it was either not visited or simply not changed. In both cases, *onChange* would not fire.

onBlur event

Related topics

The *onBlur* event always fires when a Select, Text, or TextArea component loses focus, whether or not the contents of the component have been changed. Like *onChange*, the event will not fire if the user does not visit the component at all, but if they do, you can make sure they type in something.

This type of active check has one significant drawback: Suppose the user puts the cursor in a field and then decides to abandon their entry. Because there's no way easy to tell that the user wants to leave the field to click on the Abandon button, the user will still get a validation error message. This problem can be exacerbated if the message is worded along the lines of, "The field must be filled in." The user wants to abandon the entry; why fill in the field?

In a form where the user has the option of actively abandoning any changes—as opposed to a form that the user can simply ignore—*onBlur* validation is can be too obtrusive.

onSubmit event

Related topics

Client-side JavaScript has an *onSubmit* event for the form that lets you verify all the data in a form is correct before it is submitted. However, IntraBuilder does not support it. The *onSubmit* event is not available as an event of the Form object, so you cannot automatically create a method that will exported.

Using *onSubmit* is tricky because of the way IntraBuilder-generated forms work. By default all buttons on a form are HTML submit buttons. This includes a button like the TMD Viewer form's Abandon button. When that button is clicked, the entire form is submitted, and the rowset is abandoned on the server.

Without doing some extra client-side JavaScript programming, there is no way to tell what button has been clicked in the *onSubmit* event, so you would end up doing form validation even if the user wanted to abandon changes. In fact, if the validation failed, you wouldn't let the form be submitted, so the user could never abandon their changes!

Client-server JavaScript communication

Related topics

As an example of client-side validation, the new HMN typed into the “Reset HMN” page of the login form can be checked client-side, before the form is submitted.

If the validation was a simple constant rule—for example, “can’t be below zero”—you could easily code a simple JavaScript function. But the new HMN must be within the range of message numbers currently in the Messages table. The low and high message numbers would have to be available to JavaScript running on the client browser.

Form variables, such as the minHMN and maxHMN properties used for the server-side validation, are not exported to the HTML form on the client. Text objects, such as the msgRangeLabel that also contains the low and high message numbers, are rendered as straight HTML text, and are not accessible as objects with client-side JavaScript.

Unless a visible component that’s already on the form—such as a Text or Select component—meets your needs, client-server JavaScript communication is achieved through Hidden components.

Hidden components

Related topics

A Hidden object maintains a *value*, but is not displayed on the form in the browser. It's the primary vehicle for passing values between client- and server-side JavaScript.

Even though it is not displayed on the form in the browser, as a server-side component, it has a *pageno* property, so it will be in the dynamically formulated HTML only if it is on the same page that the form is currently displaying, or if it is on page zero.

In other words, as a rule of thumb, you may want to place your Hidden objects on page zero so that they will be available no matter what page the form is displaying.

The TMD project: validating a new HMN

Related topics

To add client-side HMN validation to the TMD Login form, follow these steps:

1. Open LOGIN.JFM in the Form Designer.
2. Switch to page 6, the “reset HMN” page.
3. Drop two Hidden components on the page. Set their *name* properties to minHMN and maxHMN. These objects will replace the form variables used in The TMD project: Administering the HMN.
4. Modify the resetHMN() and setHMNButton_onServerClick() methods to use the Hidden components instead of form variables. This is done by adding a *value* property reference to minHMN and maxHMN, since they are now objects instead of simple variables, and the liberal use of the *parseInt*() function, to guarantee that the numbers are handled correctly:

```
function resetHMN()
{
  this.messages1.rowset.clearFilter();
  this.messages1.rowset.first();
  this.minHMN.value = parseInt( this.messages1.rowset.fields[ "Message
#" ].value );
  this.messages1.rowset.last();
  this.maxHMN.value = parseInt( this.messages1.rowset.fields[ "Message
#" ].value );
   this.msgRangeLabel.text = "Messages in the database are numbered from " +
                             parseInt(this.minHMN.value ) + " to " +
                             parseInt(this.maxHMN.value )
  this.pageno = 6;
  this.rowset.beginEdit();
}
function setHMNButton_onServerClick()
{
    if ( this.form.hmn.value >= 0 &&
         this.form.hmn.value <= parseInt(this.form.maxHMN.value ) ) {
    this.form.rowset.save();
    this.form.checkNewMessages();
  }
  else {
    this.form.msgRangeLabel.color = "red";
  }
}
```

5. Create the following *onChange* event handler for the hmn Text component. *onChange* is a client-side event, so this method will be exported and executed client-side:

```
function hmn_onChange()
{
  if ( parseInt( this.value ) < parseInt( this.form.minHMN.value - 1 ) ||
       parseInt( this.value ) > parseInt( this.form.maxHMN.value ) ) {
    alert( "The new HMN must be between " +
parseInt( this.form.minHMN.value - 1 ) +
           " and " + parseInt( this.form.maxHMN.value ) );
  }
}
```

   In order to see the first message, the HMN must be set to one less than the first message number, so the *onChange* event handler allows it.

6. Switch back to page 1, save the form, close the Form Designer, and run the form through a browser to test the field validation.

Using HTML frames

Related topics

IntraBuilder does not include any tools to support HTML frames, but you can create your own frameset documents and use IntraBuilder forms and reports as frame sources. With client-side JavaScript code, you can create highly interactive multi-frame database applications.

The TMD project: Viewing the new messages list

Related topics

The New Messages report created in The TMD project: Calling the New Messages report is very useful, but there is one small problem: After displaying the report, there is no way to get back to the message viewer, because reports do not go on the form stack.

By using frames, you can display the list of new messages in one frame and the message viewer form in another. Clicking a message in the report can display that message in the form. This interaction requires client-side JavaScript.

Creating the frameset document

Related topics

Frames rely on an HTML document with a set of frame tags. Unlike most HTML documents, which are intended primarily to contain content, frameset documents contain formatting information.

Here is the frameset document that will be used for the TMD project:

```
<HTML>
<HEAD>
<TITLE>Threaded Message Database</TITLE>
</HEAD>
<FRAMESET COLS="30%,*">
  <NOFRAMES>
    <H2>To use the Threaded Message Database,
    your browser needs to support frames</H2>
  </NOFRAMES>
  <FRAME NAME="reportFrame" SRC="SPLASH.HTM">
  <FRAME NAME="formFrame"   SRC="/svr/intrasrv.isv?apps/tmd/LOGIN.JFM">
</FRAMESET>
</HTML>
```

The frameset document starts with a <HEAD> and <TITLE> as usual. The <FRAMESET> tag indicates that there will be two columns, one that initially takes up 30% of the width of the browser, with the rest (indicated by the “*”) left to the remaining column.

Inside the <FRAMESET> is a <NOFRAMES> section that is ignored if the browser supports frames. If the browser does not support frames, it will ignore both the <FRAMESET> and <NOFRAMES> tags and end up displaying what’s inside the <NOFRAMES> section. It informs users that frame support is needed.

Next are the two <FRAME> tags that specify the name of the frame and its contents (or frame source). On the left is the report frame, which initially contains an HTML document with a splash screen for the TMD. On the right is the TMD Login form.

Notice that the SRC attribute for the Login form contains the complete URI needed to run the form, starting with a slash. Without the slash, the SVR directory would be relative to the TMD directory, instead of being used as an alias.

Until you have the time to create an attractive splash screen, create the following SPLASH.HTM file:

```
<HTML>
<BODY>
Splash screen goes here
</BODY>
</HTML>
```

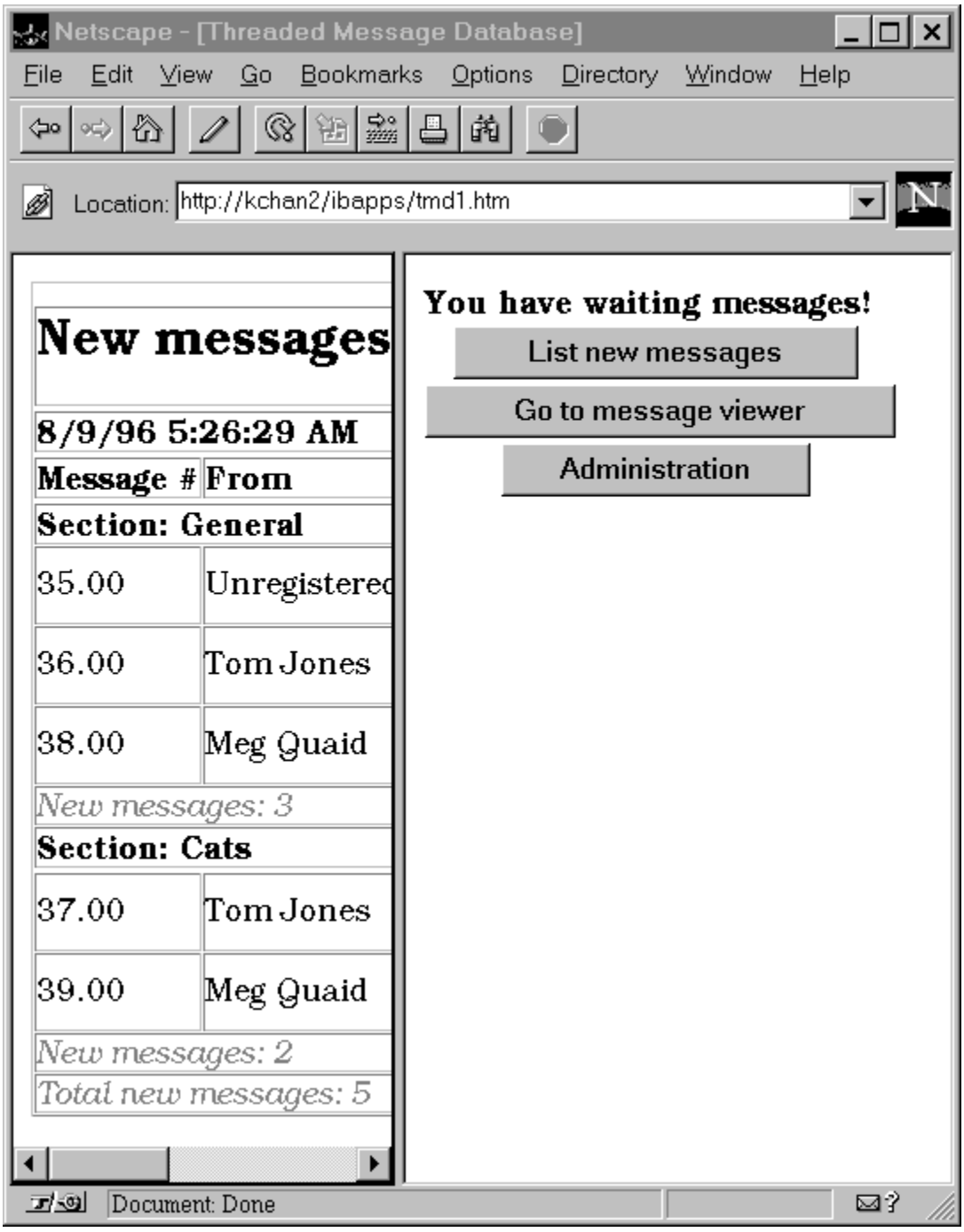
Directing output to another frame

Related topics

With the splash screen on the left, the TMD Login form works as usual on the right, asking users for their user name and password, and checking if there have been any new messages since the last time they visited.

If there are new messages, they get the page on the form that lets them display a report of the new messages, go directly to the message viewer, or administer their account.

This where the first changes must be made.



Setting the form's target

Related topics

Every form has a *target*—or destination where the response from the submitted form goes. By default this is the same window or frame that contains the form, and this has been the behavior so far. When the user clicks the button to display the New messages report, the HTML generated for the report will appear in the same frame.

By setting the form's *target* property to the reportFrame frame, the report will appear in that frame, leaving the form in the formFrame frame.

On the other hand, if the user clicks the message viewer or administration buttons, you want the form to appear in the formFrame frame. Therefore, you need to set formFrame's *target* dynamically, depending on which button is clicked. All of this must happen on the client.

onClick event

Related topics

Until now, all the buttons used in the TMD have been HTML submit buttons. Pressing a button submitted the form and ran the button's *onServerClick* event. To run code on the client, you need to use the Button control's *onClick* event instead. *onClick* event handlers are automatically exported.

Important When a button has an *onClick* event, it becomes a standard HTML button instead of a Submit button. This means that clicking it does not submit the form, and its *onServerClick* event does not fire on the server when running the form from a browser.

By using the *onClick* method, you can have the button do something on the client before the form is submitted. For the button to submit the form, its *onClick* event handler must call the form's *submit*() method. This in turn fires the form's *onServerSubmit* method on the server.

onServerSubmit event

Related topics

When a form is submitted via a Submit button, the button's *onServerClick* event fires on the server. But if the form is submitted manually via its *submit*() method, the form's *onServerSubmit* event fires instead.

There is no automatic way to tell from the *onServerSubmit* event which button or *onClick* event handler on the form caused the form to be submitted. If there is only one button on the form that submits the form manually, then it's not a problem. But in this case, there are three buttons that manually submit the form, so you will need to indicate somehow which button it was.

A Hidden object is just the component for this sort of thing.

Form sequence number

Related topics

When IntraBuilder forms are dynamically formulated as HTML, they contain a hidden component that contains a sequence number. This number is incremented every time the form is submitted and is checked to prevent the same form from being posted twice. Allowing the same form to be posted twice can cause the data that's supposed to go to one row to be posted into another row.

But in the TMD, if you first request the list of new messages, and then go to the message viewer, you are submitting the same form twice. Because the output for the report goes into the reportFrame frame, the form in the formFrame frame does not change; its sequence number is not updated. Therefore, when you try to go to the message viewer, you can't; IntraBuilder recognizes that it's the same sequence number, so you get the report again, this time in the formFrame frame.

For the multi-frame application to work, you must force IntraBuilder to accept the same form twice (or more). You should do this only if it's safe. In this case, there is no data being posted, so it doesn't matter if the form is submitted twice.

To force IntraBuilder to accept the form, set the hidden sequence number to -1. This tells IntraBuilder that you know what you're doing and that you want the form to be accepted regardless of the sequence number.

The sequence number is stored in an object named session.sequenceNumber. The name deliberately contains a dot to make it impossible for another control to have the same name. Because of the dot in the name, you must access the object through the form's *elements* array.

In Netscape Navigator, you can specify the name as a string, but in Microsoft Internet Explorer, you can't. But the sequence number is always the second element; that is, element number 1. (Unfortunately, Microsoft Internet Explorer 3.0 does not do anything with the *target* property anyway, so the report will not appear in the reportFrame frame. But by using the element number instead of the name you will avoid script errors, and the rest of the application will work.)

The TMD project: The multi-frame TMD application

Related topics

Follow these steps to enable multi-frame operation:

1 Open LOGIN.JFM in the Form Designer.

2 Drag a Hidden object from the Component Palette onto the form. Set its *pageno* to zero and its *name* property to hiddenAction.

3 Switch to page 4, the page that contains the List new messages button.

4 Right-click on the List new messages button and choose Inspector from the shortcut menu.

5 Go to the Events page of the Inspector and click the *onClick* event. Click the tool icon on the right.

6 In the Method Editor, type in the following event handler:

```
function listNewButton_onClick()
{
  this.form.target = "reportFrame";
  this.form.hiddenAction.value = "NEW";
  this.form.elements[ 1 ].value = "-1";
  this.form.submit();
}
```

7 Create the following *onClick* event handler for the Goto message viewer button:

```
function viewerButton_onClick()
{
  this.form.target = "_self";
  this.form.hiddenAction.value = "VIEWER";
  this.form.elements[ 1 ].value = "-1";
  this.form.submit();
}
```

8 Create the following *onClick* event handler for the Administration button:

```
function adminButton_onClick()
{
  this.form.target = "_self";
  this.form.hiddenAction.value = "ADMIN";
  this.form.elements[ 1 ].value = "-1";
  this.form.submit();
}
```

9 Create the following *onServerSubmit* event handler for the form:

```
function Form_onServerSubmit()
{
  if ( this.hiddenAction.value == "VIEWER" ) {
    _sys.forms.run( "VIEWER", this );
  }
  else if ( this.hiddenAction.value == "NEW" ) {
    _sys.reports.run( "NEWMSG", this );
  }
  else if ( this.hiddenAction.value == "ADMIN" ) {
    this.resetHMN();
  }
}
```

10 Switch back to page 1, save the form, and close the Form Designer.

Now, when running the form through a browser, instead of running their own *onServerClick* events on the server, each of the three buttons follows the same steps on the client:

1 Set the *target* of the form to the appropriate frame. For the List new messages button, that's the

reportFrame frame. For the other two buttons, the special frame identifier *_self* indicates that the target should be the same frame that contains the form.

2. Set the *value* of the hiddenAction component to an arbitrary string, which will be checked later.
3. Submit the form by calling its *submit*() method.

The form's *onServerSubmit* event handler then takes the appropriate action:

- When the New messages report is run, its output goes into the reportFrame frame.
- When the Viewer form is run, it appears in the formFrame frame.
- When switching the Login form to the administration page, it too appears in the formFrame frame.

Using JavaScript links

Related topics

Reports do not support server-side events the same way forms do. Interactivity is achieved solely through client-side JavaScript. You can call client-side JavaScript through HTML links embedded in the text of a report. In combination with forms in other frames, you can simulate the action of server-side events in reports.

The TMD project: Displaying a message from the new message list

Related topics

For example, here's one way to use client-side JavaScript so that clicking on a message in the message list displays that message in the formFrame frame:

1. Open NEWMSG.JRP in the Report Designer.
2. Right-click the report surface and choose Method Editor from the shortcut menu.
3. Create the new client-side method gotoMessage(). It must also set the hidden sequence number, but only if needed, because there may be data in the other frame:

```
function gotoMessage( nMsg )
{
// {Export} This comment causes this function body to be sent to the client
  var f = parent.formFrame.document.forms[0];  // Get reference to
formFrame's form
  if ( f.target == "reportFrame" ) {
    f.target = "_self";                        // Make sure the target is
the same frame
    f.elements[ 1 ].value = "-1";
  }
  f.hiddenAction.value = "VIEWER";             // Run the Viewer form
  f.hiddenMsg.value = "" + nMsg;               // Goto the desired message
#
  f.submit();                                  // Submit the form
}
```

4. Change the *text* of the "Message #" column label to just "#".
5. Select the HTML object in the report's detail band that displays the message number and change its *text* property to (all one line):

```
{||'<A HREF="javascript:gotoMessage(' +
  parseInt( this.form.messages1.rowset.fields["Message #"].value ) + ')">'
+
  parseInt( this.form.messages1.rowset.fields["Message #"].value ) +
'</A>'}
```

This self-evaluating codeblock generates an HREF link that looks like this:

```
<A HREF="javascript:gotoMessage(123)">123</A>
```

The codeblock displays the message number, and has a link to the JavaScript function gotoMessage() for each message.

The gotoMessage() function takes the message number as a parameter, sets the appropriate values in Hidden objects, and submits the form.

This means that the form in the formFrame frame must have the Hidden objects to accept the values. The Login form already contains the hiddenAction object, but it does not contain the hiddenMsg object, so:

1. Save the NEWMSG.JRP report and close the Report Designer.
2. Open LOGIN.JFM in the Form Designer.
3. Drag a Hidden object from the Component Palette. Set its *pageno* property to zero, its *name* property to hiddenMsg, and its *value* to zero.
4. Add the highlighted lines to the Form_onServerSubmit() method:

```
function Form_onServerSubmit()
{
  if ( this.hiddenAction.value == "VIEWER" ) {
    if ( parseInt( this.hiddenMsg.value ) > 0 ) {
```

```
        _sys.forms.run( "VIEWER", this, parseInt( this.hiddenMsg.value ) );
      }
      else {
        _sys.forms.run( "VIEWER", this );
      }
    }
    else if ( this.hiddenAction.value == "NEW" ) {
      _sys.reports.run( "NEWMSG", this );
    }
    else if ( this.hiddenAction.value == "ADMIN" ) {
      this.resetHMN();
    }
  }
```

5 Save the form and close the Form Designer.

Now if there is a non-zero value in the hiddenMsg object, a second parameter, the message number, is passed to VIEWER.JFM. You must modify the Viewer form to take advantage of this second parameter. Also, once the Viewer form is open, clicking on another message in the list will attempt to post values in the hiddenAction and hiddenMsg objects, so those must be created for the Viewer form.

1 Open VIEWER.JFM in the Form Designer.

2 Drag two Hidden objects from the Component Palette onto the form. Set their *pageno* property to zero, and their names to hiddenAction and hiddenMsg.

3 Add the highlighted lines to the Form_onServerLoad() method:

```
  function Form_onServerLoad()
  {
    try {
      this.userRowset = VIEWER.arguments[ 0 ].rowset;
      this.userQuery  = this.userRowset.parent;
      this.userID     = this.userRowset.fields[ "User ID" ].value;
      this.userName   = this.userRowset.fields[ "User name" ].value;
      this.HMN        = this.userRowset.fields[ "HMN" ].value;
    }
    catch ( Exception e ) {
      this.userID   = 0 ;              // Eventually will not allow access
      this.userName = "Unregistered" ; // But for now allow unregistered user
      this.HMN      = 0;
    }
    // Set "From" name on compose page to user name, because it will never
change
    this.userLabel.text = this.userName;
    this.applyHMN();
    if ( VIEWER.arguments.length == 2 ) {
      this.hiddenMsg.value = VIEWER.arguments[ 1 ];
      this.gotoMessage();
    }
    this.refreshUnlinked();
    this.sectionSelect.rowset = this.sections1.rowset;
    this.sectionSelect.field  = "Name";
  }
```

4 The *onServerLoad* event handler now calls a method named gotoMessage()—not to be confused with the client-side JavaScript function with the same name in the New messages report. Create a new method named gotoMessage():

```
  function gotoMessage()
  {
```

```
    this.rowset.applyLocate( '"Message #" = ' +
parseInt( this.hiddenMsg.value ) );
  }
```

5 The last step is to create an *onServerSubmit* event handler for the form:

```
  function Form_onServerSubmit()
  {
    if ( this.pageno == 2 ) {
      // Abandon edit if necessary
      this.abandonButton.onServerClick();
    }
    this.gotoMessage();
  }
```

6 Save the form and close the Form Designer.

Feature summary

Related topics

The New messages report is in the reportFrame frame on the left. Each message # in the report is actually a link that calls an exported JavaScript function gotoMessage().

In the report, the gotoMessage() function sets the values of the two Hidden objects which it assumes are in whatever form in the formFrame frame. It sets hiddenAction to “VIEWER” and hiddenMsg to the message number, and then submits that form by calling the form’s *submit*() method.

If the Login form is in the formFrame frame, then its *onServerSubmit* event handler— which was originally setup to handle the three buttons that set the form’s *target* on the client before running the appropriate form or report—sees the “VIEWER” in hiddenAction. It then looks for a non-zero *value* in the hiddenMsg object. If it is non-zero, it is a message number, which it passes as a second parameter to VIEWER.JFM.

When VIEWER.JFM opens, it checks to see whether two parameters were passed to it. If there were two, it takes the second parameter and saves it in its own hiddenMsg object and calls its own gotoMessage() method.

In the Viewer form, gotoMessage() takes the *value* of the hiddenMsg object and calls the rowset’s *applyLocate*() method to find the matching row. Therefore, clicking on the message number link in the New messages list when the Login form is in the formFrame frame will open the Viewer form and go directly to that message.

Once the Viewer form is open, clicking on the message number link in the New messages list has the same effect. The values of the Hidden objects are set and the form is submitted.

In the Viewer form’s *onServerSubmit*, the *value* of hiddenAction is ignored, but the object has to be there; otherwise the report’s gotoMessage() function would fail.

The Viewer form’s *onServerSubmit* event handler does two things: first it makes sure the user is not adding a new row. If they are, it abandons it by calling the Abandon button’s *onServerClick* to simulate clicking the button. Then it calls the Viewer form’s gotoMessage() method, which moves the rowset to the desired row.

Where to go from here

Related topics

If you've been following along, by now you have a moderately complex Threaded Message Database application.

The process of creating the TMD project took you through basic database operations, intermediate server-side application coding, and a touch of client-side JavaScript.

IntraBuilder provides you with the tools to create the entire spectrum of Web-based database applications; from quick and simple to complex and feature-rich. The art of IntraBuilder application development is as dynamic as the Web itself. For the latest information and discussions on IntraBuilder, be sure to investigate online resources such as Usenet and the Borland Web site.

Language definition

Related topics

The language used in IntraBuilder[a] is an extended version of the JavaScript language. JavaScript is based on Java which in turn borrows heavily from the C language, so there are some basic syntactical similarities between JavaScript and C. IntraBuilder adds full object orientation to the JavaScript language by adding formal classes. The result is a streamlined general purpose object-oriented programming language. By including classes that represent browsers, forms, reports, and databases in an advanced integrated development environment with Two-Way Tool designers, IntraBuilder enables you to develop Web-based data-centric applications with point-and-click ease.

IntraBuilder allows you to include standard JavaScript to be executed on the browser. This is known as client-side JavaScript. The extended version of JavaScript that runs on IntraBuilder is referred to as server-side JavaScript. These extensions consist primarily of more classes of objects. Except for the addition of formal classes and exception handling on the server, the structures of the two versions of JavaScript are the same.

These topics define the language elements in standard client-side JavaScript and the IntraBuilder server-side extensions. After a brief overview of basic language attributes, which is geared toward those with previous programming experience, the language is described from its most fundamental elements, data types, to the most general.

Basic attributes

Related topics

If you're familiar with another programming language, knowing the following attributes will help orient you to JavaScript. If JavaScript is your first programming language, you may not recognize some of the terminology below. Keep the rules in mind; the terminology will be explained later in this series of topics.

- JavaScript is case-sensitive.

  You must capitalize the keywords, functions, and property names exactly as they are shown in the language reference. For example, the property onClick has a lowercase "o" and an uppercase "C"; not OnClick, onclick, or any other variation.

  Rules of thumb for how things are capitalized are listed in Syntax conventions. You are encouraged to follow these rules when you create your own names for variables and properties.

- JavaScript is zero-based.

  This means that when numbering things, JavaScript starts with zero, not one. For example, the first element in an array is element zero. The first character in a string is index position number zero. The getMonth() function returns zero for January, not one.

- Each statement may end with an optional statement terminator, the semicolon.

  Some languages, like C, require that you end each statement with a special character, a statement terminator. This allows you to format the code any way you want and easily split a long statement into multiple lines, but until it becomes second nature, it's a common mistake to forget them. Other languages, like BASIC, assume there is one statement on each line. If you have a long statement, you need to use a special character, a line continuation character, at the end of a line to indicate that the next line is a continuation of the current line.

  JavaScript takes a compromise approach. You may include a statement terminator, which is the semicolon (;), just as it is in C. It doesn't hurt to include it, and if it's there, it clearly marks where the end of the statement should be. Otherwise, the end of a line is assumed to be the end of a statement, unless the statement in that line is incomplete, in which case the statement is assumed to continue on the next line. There is no line continuation character.

- There are no keyword pairs.

  Some languages have paired keywords for language structures, like IF/ENDIF and FOR/NEXT. By using paired keywords, you may include any number of statements (zero or more) in between.

  JavaScript does not use paired keywords. Instead, it expects either a single statement in the structure, or more likely, any number of statements inside a set of curly braces ({ }).

- Literal strings are delimited by either single or double quotes, and may include escape sequences.

- JavaScript is weakly typed with automatic type conversion.

  You don't have to declare a variable before you use it. You can change the type of a variable at any time.

- JavaScript's object model supports dynamic subclassing.

  Dynamic subclassing allows you to add new properties on-the-fly, properties that were not declared in the class structure.

Data types

Related topics

Data is both the means and the end for both programming and databases. Because IntraBuilder is an extended version of the JavaScript language designed to manipulate databases, there are three categories of data types:

- Simple data types common to both the scripting language and databases
- Database-specific data types
- Data types used in programming

Simple data types

Related topics

There are four simple data types common to both JavaScript and databases:

- String
- Numeric
- Logical or boolean
- Null

Keep in mind that different table formats support different data types to varying degrees.

For each of these data types, there is a way to designate a value of that type in JavaScript code. This is known as the literal representation.

String data

Related topics

A string is composed of zero or more characters: letters, digits, spaces, or special symbols. A string with no characters is called an empty string or a null string (not to be confused with the null data type).

The maximum number of characters allowed in a string depends on where that string is stored. In IntraBuilder, the maximum is approximately 2 billion characters, if you have enough virtual memory. For DBF (dBASE¨) tables, you may store 254 characters in a character field and an unlimited number in a memo field. For DB (Paradox) tables, the limit is 255 characters in an alpha field, and no limit with memo fields. Different browsers and database servers on different platforms each have their own limits.

Literal character strings must be enclosed in matching single or double quotation marks, as shown in the following examples:

```
'text'
"text"
```

A literal null string, or empty string, is indicated by two matching quotation marks with nothing in between.

To include a quotation mark inside a literal string, precede it with a backslash (the escape symbol), as in these examples:

Literal	Value
"Steve said it's \"insanely great\""	Steve said it's "insanely great"
'Bill promised it would ship in late \'93'	Bill promised it would ship in late '93

Notice that, as with "it's" in the first example, you don't have to use the escape symbol before a quote if it is not the kind of quote that is used to enclose the literal string, although it would work either way. Since the backslash is the literal string escape character, to include a backslash in a literal string, you must precede it with a backslash as well; that is, you need to have two backslashes for every one you want. This comes up particularly with file and path names, which use the backslash to indicate directories, as shown in this example statement, which changes the current directory:

```
_sys.os.changeDir( "C:\\WEBSITE\\CGI-WIN" )
```

IntraBuilder also allows forward slashes in paths. Forward slashes don't have to be preceded with the escape symbol. For example, the statement above could be written as:

```
_sys.os.changeDir( "C:/WEBSITE/CGI-WIN" )
```

You may also insert some special characters in a literal string by using one of the following backslash-and-character combinations:

Combination	Result	ASCII value
\b	backspace	8
\f	form feed	12
\n	new line	10
\r	carriage return	13
\t	tab	9

Numeric data

Related topics

JavaScript supports a single numeric data type. It does not distinguish between integers and non-integers, which are also referred to as floating-point numbers. Table formats vary in the types of numbers they store. Some support short
(16-bit) and long (32-bit) integers or currency in addition to a numeric format. When these numbers are read into IntraBuilder, they are all treated as plain numbers. When numbers are stored into tables, they are automatically truncated to fit the table format.

In JavaScript, a numeric literal may contain a fractional portion, or be multiplied by a power of 10. The following are all valid numeric literals:

- 42
- .315
- 4.6
- 5e7
- 19e+4
- 8.306E–2

As the examples show, the “E” to designate a power of 10 may be uppercase or lowercase, and you may include a plus sign to indicate a positive power of 10 even though it is unnecessary.

In addition to decimal literals, you may use octal (base 8) or hexadecimal (base 16) literal integers. If an integer starts with a zero (0), it is assumed to be octal, with digits from 0 to 7. If it starts with 0x or 0X, it is hexadecimal, with the digits from 0 to 9 and the letters A to F, uppercase or lowercase. For example,

Literal	Base	Decimal value
031	Octal	25
0x64	Hexadecimal	100

Logical data

Related topics

A logical, or boolean, value can be only one of two things: true or false. These two logical values are expressed literally in JavaScript by the reserved words true and false.

Null values

Related topics

JavaScript supports a special value represented by the reserved word null. It is its own data type, and is used to indicate a nonexistent or undefined value. A null value is different from a blank or zero value; null is the absence of a value.

The DBF (dBASE) table type does not support nulls, but most other tables, including DB (Paradox), do. A null value in a field would indicate that no data has been entered into the field, like in a new row, or that the field has been emptied on purpose. In certain summary operations, null fields are ignored. For example, if you are averaging a numeric field, rows with a null value in the field are ignored. If instead a null value was considered to be zero or some other value, it would affect the average.

Null is also used in JavaScript to indicate an empty function pointer, a property or variable that is supposed to refer to a function, but doesn’t contain anything.

Database-specific data types

Related topics

There are a number of data types supported by different databases that do not have a direct equivalent in JavaScript. The following list is not exhaustive; a new or upgraded table format may introduce new types. In any case, the type is represented by the closest matching JavaScript data type, with the string type being the catchall, since all data can be represented as a bunch of bytes.

The common database-specific types are:

- Date and date/time
- Memo
- Binary and OLE

Date and date/time data

Related topics

Dates and times are handled in JavaScript by Date objects. Objects are explained later and Date objects in particular are fully detailed in the topic Date and time.

Memo data

Related topics

As far as IntraBuilder is concerned, a memo is just a character string; potentially a very long one. For tables, it is important to distinguish between a character field, which is of fixed and usually small size, and a memo field, which is unlimited in size. For example, a character field might contain the title of a court decision, and the memo field contain the actual text of that court decision.

Binary and OLE data

Related topics

Binary and OLE data are similar to memos, except that they are usually meant to be modified by external programs, not IntraBuilder. For example, a binary field might contain a graphic bitmap, which IntraBuilder can display, but you cannot edit the bitmap with IntraBuilder.

Programming data types

Related topics

There are three data types used specifically for programming:

- Object reference
- Function pointer
- Codeblock

These types are explained later, in the context in which they are used.

Operators and symbols

Related topics

An operator is a symbol, set of symbols, or keyword that performs an operation on data. IntraBuilder provides many types of operators, used throughout the language, in the following categories:

Category	Operators
Assignment	= += –= *= /= %= <<= >>= >>>= &= ^= |=
Comparison	== > < >= <= !=
String	+
Numeric	+ – * / % ++ – –
Logical	&& || !
Bitwise	& | ^ << >> >>> ~
Object	. [] :: new
Function	()
Conditional	?:

All operators require either one or two arguments, called *operands,* with the exception of the conditional operator (?:) which requires three. Those that require a single operand are called *unary operators*; those requiring two operands are called *binary operators*. For example, the logical not operator (!) is a unary operator:

```
!endOfSet
```

The (*) is the binary operator for multiplication, for example,

```
59 * 436
```

If you see a symbol in JavaScript code, it's probably an operator, but not all symbols are operators. For example, quote marks are used to denote literal strings, but are not operators, since they do not act upon data—they are part of the representation of a data type.

Another common symbol is the end-of-line comment symbol, a double slash. It and everything on the line after it are ignored by JavaScript. For example,

```
calcAverages();              // Call the function named calcAverages
```

All operators and symbols are described in full in the Operators and Symbols section of this Help file.

Reserved words

Related topics

The core of the JavaScript language is composed of reserved words, words that have a specific meaning to JavaScript. They may be used only for their designated purpose. For example, you cannot create a variable named true since that's reserved to designate the literal logical value true. Most of these reserved words fall into three general categories:

- Literal values, such as *true*, *false*, and *null*.
- Keywords for control statements, such as *if*, *else*, *for*, and *while*.
- Structural keywords, such as *function*, *class*, and *try*.

The following words are reserved by JavaScript. Some of these words are currently implemented in the language; the rest are reserved for future use.

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
false	final	finally	float	for
function	goto	if	implements	import
in	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
super	switch	synchronized	this	throw
throws	transient	true	try	var
void	while	with		

In addition, IntraBuilder adds a few of its own reserved words, mostly to support the *extern* statement:

_sys	cdecl	clear	custom	extern
from	intdebug	pascal	quit	stdcall
unsigned				

Names

Related topics

Any word that is not a reserved word is considered a name. Names are given to variables, properties, events, methods, functions, and classes. The following rules are the naming conventions in JavaScript:

- A name begins with an underscore or letter, and contains any combination of underscores, letters, or digits.
- The letters may be uppercase or lowercase. JavaScript is case-sensitive.
- With IntraBuilder, only the first 32 characters in a name are significant. There can be more than 32, but the extra characters are ignored. For example, the following two names are considered to be the same:

```
theFirst_32_CharactersAreTheSameButTheRestArent
theFirst_32_CharactersAreTheSameAndTheRestDontMatter
```

- A reserved word cannot be used as a name.

The following are some examples of valid names:

```
x
DbException
DBException         // Different than above; second letter is uppercase
parseInt
Form
messages1_onOpen
```

Whitespace

Related topics

Spaces, tabs, and line breaks are all considered whitespace. In general, you can place as much or as little whitespace in your code as you want, with the following exceptions:

- No extra whitespace inside a literal value. A space between quotes is counted as a space. A literal string must begin and end on the same line. Whitespace in the middle of a number creates two separate numbers.
- An end-of-line comment goes as far as the end of that line. To create a multi-line comment, use a block comment, or start another end-of-line comment on the next line.

Use whitespace to make your code easier to read and follow. Indent code in control statements and functions, as shown later.

Expressions

Related topics

An expression is anything that results in a value. Expressions are built from literal data, names, and operators.

Basic expressions

Related topics

The simplest expression is a single literal data value; for example,

```
6                               // The number 6
"eloign"                        // The string "eloign"
```

You can use operators to join multiple literals; for example,

```
6 + 456 * 3                     // The number 1374
"sep" + "a" + "rat" + "e"       // The string "separate"
```

To see the value of an expression in the Script Pad, precede the expression with the ? symbol:

```
? 6 + 456 * 3                   // Displays 1374
```

Variables

Related topics

Variables are named locations in memory where you store data values: strings, numbers, logical values, nulls, object references, function pointers, and codeblocks. You assign each of these values a name so that you can later retrieve them or change them.

You can use these values to store user input, perform calculations, do comparisons, define values that are used as parameters for other statements, and much more.

Assigning variables

Related topics

Before a variable can be used, a value must be assigned to it. Use a single equal sign to assign an expression to a variable; for example,

```
alpha = 6 + 456 * 3    // alpha now contains 1374
```

An assignment is itself an expression; its value is the value that was assigned. You can assign the same value to multiple variables like this:

```
beta = gamma = delta = 23
```

In this example, delta is assigned the number 23, and the value of that assignment expression is itself 23, which is assigned to gamma. The value of that expression is also 23, which is assigned to beta. This behavior is not restricted to simple assignment. For example, now that delta contains 23, the following expression uses the += operator to add 7 to delta:

```
epsilon = delta += 7
```

The result of this assignment is 30, which is assigned to epsilon.

Using variables in expressions

Related topics

When a variable is used outside of an assignment expression, either by itself or with a non-assignment operator, its value is retrieved. For example, type the following lines in the Script Pad, without the comments:

```
alpha = 6           // Assigns 6 to alpha
beta = alpha * 4    // Assigns values of alpha (6) times 4 to beta
? beta              // Displays 24
```

Type conversion

Related topics

When combining data of two different types with operators, they must be converted to a common type. If the type conversion does not occur automatically, it must be done explicitly.

Automatic type conversion

Related topics

JavaScript features automatic type conversion between its simple data types. When a particular type is expected, either as part of an operation or because a property is of a particular type, automatic conversion may occur. In particular, both numbers and logical values are converted into strings, as shown in the following examples:

```
"There are " + 6 * 2 + " in a dozen"    // The string "There are 12 in a dozen"
"" + 4                                  // The string "4"
"2 + 2 equals 5 is " + ( 2 + 2 == 5 )   // The string "2 + 2 equals 5 is false"
```

As shown above, to convert a number into a string, simply add the number to an empty string. Be careful, though; the following expression doesn't work as you might expect:

```
"The answer is " + 12 + 1       // The string "The answer is 121"
```

The number 12 is converted to a string and concatenated, then the number 1 is converted and concatenated, yielding "121". To concatenate the sum of 12 plus 1, use parentheses to force the addition to be performed first:

```
"The answer is " + (12 + 1)     // The string "The answer is 13"
```

Explicit type conversion

Related topics

In addition to automatic type conversion, there are a number of functions to convert from one type to another:

- String or floating point to integer: use the *parseInt*() function:

```
parseInt( "4" )        // The number 4
parseInt( 5.6 )        // Truncates decimals to yield the number 5
parseInt( "7" ) + 8    // The number 15
```

- String to floating point number: use the *parseFloat*() function:

```
parseFloat( "5.6" )    // The number 5.6
```

- String containing any type of expression to the value of that expression: use the *eval*() function:

```
eval( "4" )            // The number 4
eval( "4 + 2" )        // The number 6
eval( "true" )         // The logical value true
```

Arrays

Related topics

IntraBuilder supports a rich set of array classes. An array is an n-dimensional list of values stored in memory. Each entry in the array is called an element, and each element in an array can be treated like a variable.

To create an array, you can use the object syntax detailed in Array objects, but for a one-dimensional array, you can also use the literal array syntax.

Literal arrays

Related topics

A literal array declares and populates an array in a single expression. For example,

```
aTest = { 4, "yclept", true }
```

creates an array with three elements:

- The number 4
- The string “yclept”
- The logical value *true*

and assigns it to the variable aTest. The three elements are enclosed in curly braces and separated by commas.

Array elements are referenced with the index operator, the square brackets ([ ]). Elements are numbered from zero. For example, the third element is element number 2:

```
? aTest[ 2 ]         // Displays true
```

You can assign a new value directly to an element, just like a variable:

```
aTest[ 2 ] = false  // Element now contains false
```

The curly braces are also used for control statements, functions, codeblocks, and classes, all explained later.

Complex expressions

Related topics

The following is an example of a complex expression that uses multiple names, operators, and literal data. It is preceded by a question mark so that when it's typed into the Script Pad, it displays the resulting value:

```
? {"1st","2nd","3rd","4th"}[ new Date().getMonth() / 3 ] + " quarter"
```

Except for the question mark, the entire line is a single complex expression, made up of many smaller basic expressions. The expression is evaluated as follows:

- A literal array of literal strings is enclosed in braces, separated by commas. The strings are enclosed in double quotation marks.
- The resulting array is referenced using the square brackets as the index operator. Inside the square brackets is a numeric expression.
- The numeric expression begins with an object created by the operator *new*, the class name Date, and the parentheses which act as the call operator. This new object contains the current date and time.
- The resulting object is referenced with the dot operator. Its method *getMonth*() is called with the call operator.
- The method call gets the month from the date, which is then divided by 3. The result is the number that is used to index the array. Array indexes are always integers, so any fractional portion is truncated.
- The string containing the ordinal number for the calendar quarter that corresponds to the month of the current date is extracted from the array, which is then added to the literal string "quarter".

The value of this complex expression is a string like "4th quarter".

Statements

Related topics

A statement is an instruction that directs IntraBuilder to perfom a single action. This action may be simple or it may be complex, causing other actions to occur. You may type and execute individual statements in the Script Pad.

Basic statements

Related topics

The simplest statement is an expression by itself, for example,

```
6;                  // The number 6
6 + 7;              // Calculate 6 plus 7
new Date();         // Create new Date object
bakers = 6 + 7;     // Assign variable
f.open();           // Call the open() method of the object f
```

The first two examples are legal, but useless. In the first one, the expression 6 is evaluated—the result is the number 6—and discarded. In the second example, the result of the expression 6 + 7 is calculated, and again, since nothing is done with the result, it is discarded.

The third example is almost as useless. A new Date object is created, which causes IntraBuilder to get the current date and time, allocate memory for the object, and actually create the object. But since the reference to the newly created object is not stored anywhere, the object is immediately destroyed.

The fourth and fifth examples are expressions that actually do something. The fourth expression is an assignment. IntraBuilder calculates 6 + 7, but this time the result is assigned to the variable bakers.

The fifth example takes two names and two operators: the object referenced by the variable f—suppose it's a Form object—and that object's open method. The dot operator is used to access the method of the object, and the call operator (the parentheses) actually calls, or executes, that method.

Statement terminator

Related topics

Note that each example statement ends with a semicolon, JavaScript's statement terminator character. While statement terminators are not strictly required, it is recommended that you use them. A statement terminator removes ambiguity, clearly indicating the end of a statement.

Control statements

Related topics

IntraBuilder supports a number of control statements that can affect the execution of other statements. Control statements begin with a reserved word, and fall into the following categories:

- Conditional execution
- *if*
- *switch*
- Looping
- *for*
- *while*
- Object manipulation
- *for...in*
- *with*
- Exception handling
- *try*

These control statements are fully documented in the <~Core language~idh_intro_corelangJMP~> topic series. Most of the control statements can control either a single statement—including another control statement—or a block of statements inside curly braces. The exceptions are *switch* and *try*, which require the curly braces.

Even a single statement may be included inside curly braces. For example, suppose you use an *if* statement to add a value to a total, but only if the value is greater than zero. The *if* statement may appear in a number of ways:

```
if ( n > 0 ) total += n;
if ( n > 0 )
  total += n;
if ( n > 0 ) {
  total += n;
}
if ( n > 0 )
{
  total += n;
}
```

Because whitespace, including line breaks, don't matter, the first two examples are syntactically equivalent, as are the second two. The only real difference is whether the single statement is included inside the curly braces.

Always using the braces has the advantage of clearly indicating which statements are under the control of the control statement. Also, adding additional statements is easy; you don't have to go back an add the braces. Finally, braces eliminate any confusion over nested control statements. For example, when does the *else* statement execute in the following nested control statement?

```
if ( n > 0 ) if ( total < 100 ) total += n; else total = n;
```

Indenting will help someone reading the code know what you meant to do, either this:

```
if ( n > 0 )
  if ( total < 100 )
    total += n;
  else
    total = n;         // If n > 0 and total >= 100
```

or this:

```
if ( n > 0 )
  if ( total < 100 )
    total += n;
else
  total = n;           // If n <= 0
```

But because whitespace is ignored in JavaScript, indenting doesn't mean anything to the compiler. The semicolons don't help either, since they are optional to begin with.

It turns out that the first interpretation is how the code is compiled. Using braces, it would have been clear, either this:

```
if ( n > 0 ) {
  if ( total < 100 ) {
    total += n;
  }
  else {
    total = n;         // If n > 0 and total >= 100
  }
}
```

or this:

```
if ( n > 0 ) {
  if ( total < 100 ) {
    total += n;
  }
}
else {
  total = n;           // If n <= 0
}
```

Functions and codeblocks

Related topics

A function is a code module—a set of statements—to which a name is assigned. The statements can be called by the function name as often as needed. Functions also provide a mechanism whereby the function can take one or more parameters that are acted upon by the function.

A function is called by following the function name with a set of parentheses, which act as the call operator. When discussing a function, the parentheses are included to help distinguish functions from other language elements like variables.

For example, the function isBlank() receives the character string parameter cArg and returns a logical value to indicate whether the string is blank.

```
function isBlank( cArg ) {
  var c = cArg;                        // Make a work copy
  while ( c.length > 0 && c.charAt( 0 ) == " " ) {
    c = c.substring( 1, c.length );  // Remove first character (a space)
  }
  return c == "";
}
```

isBlank() works by removing all the spaces from the parameter string, and if there's nothing left, the string was blank.

Functions are identified by the reserved word *function* in a script file; they cannot be typed into the Script Pad. While many functions use *return* to return a value, they are not required to do so. If a function does not explicitly return a value, either because the *return* statement does not include a value or the last statement in the function has been executed, the result of the function call is the literal value *null*.

Function parameters

Related topics

There are two ways to pass parameters, or arguments, to functions:

- Through formal parameters
- The function's *arguments* array

Formal function parameters

Related topics

Formal parameters are variables that are declared between the parentheses after the function name in the function definition, like the variable cArg in the isBlank() function:

```
function isBlank( cArg ) {
```

When calling the function, each expression passed to the function is copied into the corresponding parameter variable for use in the function.

Formal parameters are passed by reference.

The arguments array

Related topics

Whenever a function is called, an object with the same name as the function is created. It contains a property named *arguments* that points to an array. This array has a *length* property that indicates the number of arguments passed to the function, and each argument is stored as an element of the array.

This allows a function to take a variable number of parameters. For example, this function will calculate the mean average of all the numbers passed to it:

```
function averageOf()
{
  var nTotal = 0;
  for ( var n = 0; n < averageOf.arguments.length; n++ ) {
    nTotal += averageOf.arguments[ n ];
  }
  return nTotal / averageOf.arguments.length;
}
```

This function uses a *for* loop to add all the numbers passed to it. Notice the use of the *length* and elements of the *arguments* array, which is a property of an object with the same name as the function.

The *arguments* array is always created, even if a function has formal parameters. The contents of the *arguments* array is read-only.

Function pointers

Related topics

The name of a function is actually a pointer to that function. Applying the call operator to a function pointer calls that function.

Function pointers are a distinct data type, and can be assigned to other variables or passed as parameters. The function can then be called through that function pointer variable. Here is an example of using the built-in functions *parseInt*() and *parseFloat*() in the Script Pad:

```
x = parseInt   // No parentheses, function name only
? x( 5.67 )    // Calls paraseInt(), displays 5
x = parseFloat
? x( 5.67 )    // Calls parseFloat(), displays 5.67
```

Function pointers enable you to assign a particular function to a variable or property. The decision can be made up front and changed as needed. Then that function can be called as needed, without having to decide which function to call every time.

Codeblocks

Related topics

While a function pointer points to a function defined in a script, a codeblock is compiled code that can be stored in a variable or property. Codeblocks do not require a separate script; they actually contain code. Codeblocks are another distinct data type that can be stored in variables or properties and passed as parameters, just like function pointers.

Codeblocks are called with the same call operator that functions use, and may receive parameters. Because codeblocks have no name, all parameters must be formal.

There are two types of codeblocks:

- Expression codeblocks
- Statement codeblocks

Expression codeblocks return the value of a single expression. Statement codeblocks act like functions; they contain one or more statements, and may return a value.

In terms of syntax, both kinds of codeblocks are enclosed in curly braces ({ }) and

- Cannot span multiple lines.
- Must start with either two pipe characters (||) or a semicolon (;)
- If ; it must be a statement codeblock with no parameters
- If || it may be either an expression or statement codeblock
- The || are used for parameters to the codeblock, which are placed between the two pipe characters. They may also have nothing in-between, meaning no parameters for either an expression or statement codeblock.
- Parameters inside the ||, if any, are separated by commas.
- For an expression codeblock, the || must be followed by one and only one expression, with no ;

These are valid expression codeblocks:

```
{|| false}
{|| new Date()}
{|x| x * x}
```

- Otherwise, it is a statement codeblock. A statement codeblock may begin with || (again, with or without parameters in-between).
- The first statement in a statement codeblock must be preceded by a ; Statements may be terminated with semicolons for clarity, just like in a script. The usual convention is to terminate all the statements except that last. These are valid statement codeblocks (the first two are functionally the same):

```
{; _sys.scriptOut.clear()}
{||; _sys.scriptOut.clear()}
{|x|; _sys.scriptOut.writeln( x )}
{|x|; _sys.scriptOut.clear(); _sys.scriptOut.writeln( x )}
```

- You may use a *return* inside a statement codeblock, just like with any other function. (A *return* is implied with an expression codeblock.) For example,

```
{|o,p|; for ( c in o ) { if ( c == p ) { return true; } } return false}
```

Because codeblocks don't rely on functions in scripts, you can create them in the Script Pad. For example,

```
square = {|x| x * x}  // Expression codeblock
? square( 4 )         // Displays 16
so = {|x|; _sys.scriptOut.writeln( x )}  // Statement codeblock
so( square( 5 ))      // Displays 25
isProperty = {|o,p|; for ( c in o ) { if ( c == p ) { return true; } } return
false}
? isProperty( _sys, "script" )     // Displays false
? isProperty( _sys, "scriptOut" )  // Displays true
```

Codeblocks vs. functions

Related topics

A codeblock is a convenient way to create a small anonymous function and assign it directly to a variable or property. The code is physically close to its usage and easy to see. In contrast, a function pointer refers to a function defined elsewhere, perhaps much later in the same script file, or in a different script file.

Functions are easier to maintain. Their syntax is not cramped like codeblocks, and it's easier to include readable comments in the code. In a class definition, all *function* definitions are all together at the bottom. Codeblocks are scattered throughout the constructor. If you want to run the same code from multiple locations, using function pointers that point to the same function means that changing the code requires changing the function once; multiple codeblocks would require changing each codeblock individually.

You can create a codeblock at run time by constructing a string that looks like a codeblock and using the *eval*() function to evaluate it.

Finally, client-side JavaScript does not support codeblocks.

Objects and classes

Related topics

An object is a collection of properties. Each of these properties has a name. These properties may be simple data values, such as numbers or strings, or references to code, such as function pointers and codeblocks. A property that references code is called a method. A method that is called by IntraBuilder in response to a user action is called an event.

Objects are used to represent abstract programming constructs, like arrays and files, and visual components, like buttons and forms. All objects are initially based on a class, which acts as a template for the object. For example, the Button class contains properties that describe the position of the button, the text that appears on the button, and what the button should do when it is clicked. All these properties have default values. Individual button objects are instances of the Button class that have different values for the properties of the button.

IntraBuilder contains many built-in, or stock, classes, which are documented throughout the *Language Reference*. You can extend these stock classes or build your own from scratch with a new *class* definition.

While the class acts as a formal definition of an object, you can always add properties as needed. This is called dynamic subclassing.

Dynamic subclassing

Related topics

To demonstrate dynamic subclassing, start with the simplest object: an instance of the Object class. The Object class has no properties. To create an object, use the *new* operator, along with the class name and the call operator, which would include any parameters for the class (none are used for the Object class).

```
obj = new Object()
```

This statement creates a new instance of the Object class and assigns an object reference to the variable obj. Unlike variables that contain simple data types, which actually contain the value, an object reference variable contains only a reference to the object, not the object itself. This also means that making a copy of the variable:

```
copy = obj
```

does not duplicate the object. Instead, you now have two variables that refer to the same object.

To assign values to properties, use the dot operator. For example,

```
obj.name = "triangle"
obj.sides = 3
obj.length = 4
```

If the property does not exist, it is added; otherwise, the value of the property is simply reassigned. This behavior can cause simple bugs in your scripts. If you mistype a property name during an assignment, for example,

```
obj.Sides = 4   // should not be capital S
```

a new property is created instead of changing the value of the existing property you intended.

Methods

Related topics

A method is a function or codeblock assigned to a property. The method is then called through the object via the dot and call operators. Continuing the example above:

```
obj.perimeter = {|| this.sides * this.length}
? obj.perimeter()    // Displays 12
```

As you may have deduced by now, the object referred to by the variable obj represents a regular polygon. The perimeter of such a polygon is the product of the length of each side and the number of sides.

The reference *this* is used to access these values. In the method of an object, the reference *this* always refers to the object that called the method. By using *this*, you can write code that can be shared by different objects, and even different classes, as long as the property names are the same.

A simple class

Related topics

Here is a class representing the polygon:

```
class RegPolygon
{
  this.sides  = 3;   // Default number of sides
  this.length = 1;   // and default length
  function perimeter()
  {
    return this.sides * this.length;
  }
}
```

The top of the *class* definition, up to the first *function*, is called the class constructor, which is executed when an instance of the class is created. In the constructor, the reference *this* refers to the object being created. The sides and length properties are added, just as they were before.

The function in the class definition is considered a method, and the object automatically has a property with the same name as the method that points to the method. The code is the same, but now instead of a codeblock, the method is a function in the class. Methods have the advantage of being easier to maintain and subclass.

Scripts

Related topics

A script contains any combination of the following items:

- Statements to be executed
- Functions and classes that may be called
- Comments

The JavaScript compiler in IntraBuilder also supports a standard language preprocessor, so a script that is run by IntraBuilder may contain preprocessor directives. These directives are not part of the JavaScript language; instead they form a separate simple language that can affect the code compilation process, and are explained later.

Script files

Related topics

A script file may have any file-name extension, although there are a number of defaults:

- A script containing a form is .JFM
- A script containing a report is .JRP
- Any other script is .JS

These file-name extensions are assumed by the IntraBuilder Explorer, the Script Editor, and the methods of the *_sys* object.

When a script is compiled into byte code by IntraBuilder, it stores the byte code in a file with the same name and extension, but it changes the last character of the extension to the letter “O”: .JS becomes .JO, .JFM becomes .JFO, and .JRP becomes .JRO.

Script execution

Related topics

You may execute a script by calling any of the run() methods of the *_sys* object: scripts.run(), forms.run(), or reports.run(), each of which assume their own default file-name extension. If you run the script through the IntraBuilder Explorer, the equivalent *_sys* method will be streamed out to the Script Pad and executed. You can also call a .JS script by name with the call operator, the parentheses, in the Script Pad; for example,

```
sales_report()
```

will attempt to execute the file SALES_REPORTS.JS. Since the operating system is not case-sensitive about file names when searching for files, neither is IntraBuilder.

When a script is executed, an *arguments* array is created, just as it is for a function. The *arguments* array is a property of an object with the same name as the script in uppercase. For example, the *arguments* array for GuestBook.js would be GUESTBOOK.*arguments*.

A basic script simply contains a number of JavaScript statements, which are executed once in the order that they appear in the script file, from the top down. For example, the following four statements remember the current directory, switch to another directory, execute a report, and switch back to the previous directory:

```
var cDir = _sys.os.changeDir();
_sys.os.changeDir( "C:\\SALES" );
_sys.reports.run( "DAILY" );
_sys.os.changeDir( cDir );
```

Control statements are acted upon as they occur; they may affect the execution of the code that they contain. Some statements may be executed only when a certain condition is true and other statements may be executed more than once in a loop. But even within these control statements, the execution is still basically the same, from the top down.

When and if there are no more statement to execute, the script ends, and control returns to where the script was called. For example, if the script was executed from the Script Pad, then control returns to the Script Pad and you can do something else.

Functions and classes

Related topics

Functions and classes affect execution in two ways. First, when a function or class definition is encountered in the straight top-down execution of a script, it is simply skipped over; the statements contained within are not executed. This means that you can place functions and classes before the other statements in the script, after them, or interspersed throughout. Usually, functions and classes are all grouped together at the end of a script, after the other statements. This makes it clear that the statements at the beginning are the ones that are executed, and once you get to a function or class, there's nothing else to do.

The second effect is that when a function, class constructor, or method is called, execution jumps into that function or class, executes that code in the usual top-down fashion, then goes back to where the call was made and continues where it left off.

Comments

Related topics

Use comments to include notes to yourself or others. The contents of a comment do not follow any JavaScript rules; include anything you want. Comments are stripped out at the beginning of the script compilation process and are ignored when scripts are executed on a browser.

A script will typically contain a group of comments at the beginning of the file, containing information like the name of the script, who wrote it and when, version information, and instructions for using it. But the most important use for comments is in the code itself, to explain the code—not obvious things like this:

```
n++    // Add one to the variable n
```

(unless you're writing example code to explain a language) but rather things like what you're doing in the overall scheme of the program, or why you decided to do something in a particular way. Decisions that are obvious to you when you write a statement will often completely bewilder you a few months later. Write comments so that they can be read by others, and put them in as you code, since there's rarely time to add them in after you're done, and you may have forgotten what you did by then anyway.

There are two types of comments: end-of-line comments and block comments.

End-of-line comments start with the two forward slashes together, with no space in between (//). Anything after them to the end of that line is considered a comment and completely ignored by JavaScript. There doesn't have to be anything on the line before the comment; in that case, the entire line is a comment.

Block comments start with a slash and an asterisk together (/*) and end with the reverse (*/). Everything between them is considered a comment and ignored. Block comments are usually used for multi-line comments; for example,

```
/*
    Sales_Report.js
    Generates daily sales report
*/
```

Or, instead, you could use an end-of-line comment for each line, which some people prefer:

```
//
//  Sales_Report.js
//  Generates daily sales report
//
```

You can do all sorts of fancy things to make the comments stand out; for example,

```
/*******************************\
*  Sales_Report.js              *
*  Generates daily sales report  *
\*******************************/
```

which is a block comment that begins with a /* and ends with a */.

Block comments can also be used to place comments in the middle of a line. For example,

```
for ( var n = 2 /* skip first */; n <= count; n += 2 /* hit every other one
*/ ) {
  Ä
}
```

Preprocessor directives

Related topics

Because preprocessor directives are not part of the JavaScript language, you cannot execute them in the Script Pad.

A preprocessor directive must be on its own line, and starts with the number sign (#). For more information about using preprocessor directives, see Preprocessor.

A simple script

Related topics

Here is a simple script that creates an instance of the RegPolygon class, changes the length of a side, and displays the perimeter:

```
/*******************************\
*  polygon.js                   *
*  A simple script demonstration *
\*******************************/
var poly = new RegPolygon();
poly.length = 4;
_sys.scriptOut.writeln( poly.perimeter() );  // Displays 12
class RegPolygon
{
  this.sides  = 3;   // Default number of sides
  this.length = 1;   // and default length
function perimeter()
  {
    return this.sides * this.length;
  }
}
```

Syntax conventions

Related topics

The *Language Reference* uses specific symbols and conventions in presenting the syntax of JavaScript language elements.

This section explains IntraBuilder syntax notation and provides an example of the various elements of the language syntax.

Syntax notation

Related topics

JavaScript statements, methods, and functions are described with syntax diagrams. These syntax diagrams consist of a least one fixed language element—the one being documented—and may include arguments, which are enclosed in angle brackets (< >).

JavaScript is case-sensitive. The fixed elements in the syntax diagram must be typed as shown. The names of the arguments are descriptive only; how the names are capitalized is irrelevant.

The following table describes the symbols used in syntax:

Symbol	Description
< >	Indicates an argument that you must supply
[ ]	Indicates an optional item
|	Indicates two or more mutually exclusive options
...	Indicates an item that may be repeated any number of times

Arguments are often expressions of a particular type. The description of an expression argument will indicate the type of argument expected, as listed in the following table:

Descriptor	Type
expC	A character expression
expN	A numeric expression
expL	A logical or boolean expression; that is, one that evaluates to *true* or *false*
exp	An expression of any type
oRef	An object reference

All the arguments and optional elements are described in the syntax description.

Syntax example

Related topics

The syntax for the *extern* statement illustrates all of the syntax symbols:

extern [*cdecl* | *pascal* | *stdcall*] <return type> <function name>
([<parameter type> [, <parameter type> ...]])
<filename expC>

- The word *extern* is a fixed language element, and must be typed in lowercase as shown.
- The calling convention, [*cdecl* | *pascal* | *stdcall*], is optional, as indicated by the square brackets.
- If you do specify a calling convention, the pipe character (|) between the three options indicates that you may choose only one of the three.
- The <return type> is a required argument. (The description of the *extern* statement includes a list of valid return types.)
- The <function name> is also a required argument.
- Following the <function name> is a set of parentheses. These are fixed language elements.
- Inside the parentheses are optional <parameter type> arguments, as indicated by the square brackets.
- The location of the comma inside the second square bracket indicates that the comma is needed only if more than one <parameter type> is specified.
- The ellipsis (...) at the end means that any number of parameter type arguments may be specified (each separated with a comma).
- After the parentheses is the required <filename expC> argument. Unlike the <return type> or <function name>, this is an expression—a character expression. It could be a literal character string, a variable that contains a character string, or any other character expression.

A simple *extern* statement with neither of the two optional elements would look like this:

```
extern int angelsOnAPin() "ANSWER.DLL"
```

The <return type> argument is *int*, and the <function name> is angelsOnAPin. A more complicated *extern* statement with a calling convention and parameters would look like this:

```
extern pascal long wordCount( char *, boolean ) cUtilDLL
```

In this example, cUtilDLL is a variable that contains the file name.

Capitalization guidelines

Related topics

JavaScript is a case-sensitive language. The following guidelines describe the standard capitalization of various language elements. You are encouraged to follow these guidelines in your own scripts.

- Reserved words are all lowercase. They also appear italicized in the *Language Reference*.
- Class names start with a capital letter. Multiple-word class names are joined together without any separators between the words, and each word starts with a capital letter. For example,

  Form
  PageTemplate
  TextArea
- Property, event, and method names start with a lowercase letter. If they are multiple-word names, the words are joined together without any separators between the words, and each word (except the first) starts with a capital letter. They also appear italicized in the *Language Reference*. For example,

  color
  dataLink
  onImageServerClick
- Variable and function names are capitalized like property names.
- Manifest constants created with the #define preprocessor directive are all uppercase, with underscores between words. For example,

  ARRAY_DIR_NAME
  NUM_REPS

Operators and symbols

Related topics

An operator is a symbol, set of symbols, or keyword that specifies an operation to be performed on data. Data is supplied in the form of arguments, or *operands*.

For example, in the expression “total = 0”, the equal sign is the operator and “total” and “0” are the operands. In this expression, the numeric operator “=” takes two operands, which makes it a *binary* operator. Operators that require just one operand (such as the numeric increment operator “++”) are known as *unary* operators. An operator that requires three operands is a *ternary* operator. IntraBuilder’s extended JavaScript language uses only one ternary operator, the conditional “?:”.

Operators are categorized by type. IntraBuilder’s operators are classified as follows:

Operator symbols	Operator category
= += –= *= /= %= <<= >>= >>>= &= ^= |=	Assignment
== > < >= <= !=	Comparison
+	String
+ – * / % ++ – –	Numeric
&& || !	Logical
& | ^ << >> >>> ~	Bitwise
. [] new ::	Object
()	Function
?:	Conditional

Most symbols you see in IntraBuilder code are operators, but not all. Quotation marks, for example, are used to denote literal strings and thus are part of the representation of a data type. Since they don’t act upon data, they’re a “non-operational” symbol.

You can use the following non-operational symbols in IntraBuilder code:

Symbols	Name/meaning
;	Statement terminator
//	End-of-line comment
/* */	Block comment
?	Script Pad writeln
{} {;} {||}	Program/literal array/codeblock markers
"" "	Literal strings
\	String escape character
#	Preprocessor directive

Operator precedence

Related topics

IntraBuilder applies strict rules of precedence to compound expressions. In expressions that contain multiple operations, parenthetical groupings are evaluated first, with nested groupings evaluated from the “innermost” grouping outward. After all parenthetical groupings are evaluated, the rest of the expression is evaluated according to the following operator precedence:

Order of precedence (highest to lowest)	Operator description or category
(*expression*)	Parenthetical grouping, all expressions
() [] . *new* ::	Object operators: call; member (square bracket or dot); *new*; scope resolution
! ~ – ++ – –	Negation, increment/decrement
* / %	Multiply, divide, modulus
+ –	Addition, subtraction
<< >> >>>	Bitwise shift
< <= > >=	Comparison
== !=	Equality
&	Bitwise And
^	Bitwise XOr
|	Bitwise Or
&&	Logical And
||	Logical Or
?:	Conditional
= += –= *= /= %= <<= >>= >>>= &= ^= |=	Assignment
,	Comma

In compound expressions that contain operators from the same precedence level, evaluation is conducted on a literal left-to-right basis, except for assignment operators, which associate from right-to-left. For example, no operator precedence is applied in the expressions 21/7*3 and 3*21/7 (both return 9).

Here’s another example:

```
4+5*(6+2*(8-4)-9)%19>=11?"yes":"no"
```

This example is evaluated in the following order:

```
8-4=4
2*4=8
6+8=14
14-9=5
5*5=25
25%19=6
4+6=10
```

A conditional operator is then applied to return the string value “yes” if the preceding expression evaluates to more than or equal to 11. If not (as in this case), the result is “no”.

In this assignment expression,

```
a = b = 2+3
```

2 and 3 are added first, yielding 5, then 5 is assigned to the variable b. The result of this assignment is the value being assigned, which is again 5, and this is assigned to the variable a.

Assignment operators

Related topics

Arithmetic assignment operators: = += –= *= /= %=

Bitwise assignment operators: <<= >>= >>>= &= ^= |=

Syntax

```
x = n
y = x
x += y
x <<= y
```

Description

Assignment operators are binary operators that assign the value of the operand on the right to the operand on the left. Unlike other operators, assignment operators associate from right to left.

The standard assignment operator is the equal sign. For example, *x* = 4 assigns the value 4 to the variable *x*, and *y* = *x* assigns the value of the variable *x* (which must already have an assigned value) to the variable *y*.

The other arithmetic assignment operators are shortcuts to extended arithmetic operations. The expression *x* += *y* means that *x* is assigned its own value plus that of *y* (*x* = *x* + *y*). Both operands must already have assigned values, or an error results. Thus, if the operand *x* has already been assigned the value 4 and
y has been assigned the value *6*, the expression *x* += *y* returns 10.

Bitwise assignment operators work the same way as the extended arithmetic operators, but assign bitwise, rather than arithmetic, calculations to the left operand.

Bitwise operators

Related topics

Bitwise OR operator: |

Bitwise XOR operator: ^

Bitwise AND operator: &

Bitwise NOT operator: ~

Bitwise shift operators: << >> >>>

Syntax

n | m
n ^ m
n & m
~ n
n << m

Description

These special types of operators evaluate decimal, hexadecimal or octal operands in their 32-bit binary form—as a set of 32 zeroes and ones—and return a value of the same type as the operand.

The bitwise OR operator, one of three binary bitwise logical operators, does a bit-by-bit comparison of the two operands. It then assigns 1 to the current bit position if either bit is 1. For example, the bitwise representation of 40 and 100 are, respectively,

00101000

01100100

Placing a 1 in the position where either bit is 1, the resulting bitwise value is

01101100

or, in decimal terms, 108. Thus, the expression 40|100 (or 100|40) returns 108.

The bitwise AND operator only places a 1 in positions where *both* bits are 1. Thus, the binary result of 40&100 is

00100000

or, in decimal terms, 32.

The bitwise XOR operator places a 1 in positions where *one* but *not both* bits are 1. Thus, the binary result of 40^100 is

01001100

or, in decimal terms, 76.

The bitwise NOT operator (~) is a unary operator that reverses all bits in the supplied operand. In effect, it takes an existing value, negates it and then negatively increments or decrements the value by one. For example, if *x* = 76 and the bitwise NOT operator is applied to the variable (~x;), the bitwise operation looks like this

00000000000000000000000001001100

11111111111111111111111110110011

The return value, in decimal terms, is –77. Note that this value results from the exact reversing of all bits in the binary representation of 76. This inversion always results in a negation and a negated increment or decrement of the supplied value. Thus, ~–45 returns 44, ~99999 returns –100000, and so on.

Bitwise shift operators work differently from their logical counterparts. They also take two operands, but the second operand is used to shift the bits left or right. Shifting one place to the left is equivalent to multiplying by 2. Shifting to the right is equivalent to dividing by 2, dropping any fractions. If you shift the bits of the value 76 two bits to the left, the binary result is

0100110000

or, in decimal terms, 304. The JavaScript expression for this shift is 76<<2.

If 76 is shifted two bits to the right with the expression 76>>2, the last two zeroes are trimmed off and the binary result is

010011

or, in decimal terms, 19.

Both of the standard right and left bit shift operations preserve signs. That is, if –76 were used in the example above, the result would be –19, as illustrated here with the full 32-bit representation of –76

11111111111111111111111110110100

and –19

11111111111111111111111111101101

Note that the two rightmost zeroes on the binary –76 are discarded to result in the binary –19.

The third type of bitwise logical operator—the zero-fill right shift operator—is used when the number being shifted is being treated as a 32-bit unsigned number. It also discards the specified number of bits from the right, but instead of preserving the left-most bit, it always inserts zeroes. For example, the 32-bit signed representation of –76 is equivalent to 4294967220 unsigned. Thus, instead of –19, the expression –76>>>2 returns

00111111111111111111111111101101

or, in decimal terms, 1073741805, which is the unsigned value divided by 4.

+ operator

Related topics Example

Binary addition, concatenation operator.

Syntax

"str1" + "str2"
n + *m*
"str" + *n*
n + *"str"*

Description

The “plus” symbol adds two numeric values or concatenates two strings. You can also use it to return a string concatenation of a number and a string.

+ operator, examples

// addition, concatenation operator examples

```
"this &" + " that"        // = this & that
5 + 5                     // = 10
"this & " + 5 + " more"   // = this & 5 more
5 + "-5"                  // = 5-5
```

Numeric operators

Related topics

Binary numeric operators: + – * / %

Unary numeric operators: – ++ – –

Syntax

n + *m*
n++
n – –
++n

Description

Perform standard arithmetic operations on two operands, or increment, decrement or negate a single operand.

All of these operators take numeric values as operands. The + (plus) symbol can also be used to concatenate strings or concatenate numeric and string values to return strings.

The – (minus) symbol can be used as either a binary or unary operator. As a unary operator, it simply returns a negated value for the operand or expression to which it is applied. For example, –(6 + 4) returns –10, –(–6 + 4) returns 2 and
–(–6 – 4) returns 10.

As binary numeric operators, the +, –, *, and / symbols perform the standard arithmetic operations addition, subtraction, multiplication and division.

The modulus operator returns the remainder of an integral division operation on its two operands. For example, 50%8 returns 2, which is the remainder after dividing 50 by 8.

The increment/decrement operators ++ and – – take a variable or property and increase or decrease its value by one. The operator may be used before the variable or property as a prefix operator, or afterward as postfix operator. For example,

```
n = 5     // Start with 5
? n++     // Get value (5), then increment
? n       // Now 6
? ++n     // Increment first, then get value (7)
? n       // Still 7
```

If the value is not used immediately, it doesn't matter whether the ++/– – operator is prefix or postfix, but the convention is postfix.

Logical operators

Related topics

Binary logical operators: && ||

Unary logical operator: !

Syntax

n && *m*
n || m
!n

Description

The && (AND) and || (OR) logical operators return a logical value (*true* or *false*) based on the result of a comparison of two operands. In a logical AND, both expressions must be *true* for the result to be *true*. In a logical OR, if either expression is *true*, or both are *true*, the result is *true*; if both expressions are *false*, the result is *false*.

When IntraBuilder evaluates an expression involving && or ||, it uses short-circuit evaluation:

- *false* && <any expL> is always *false*
- *true* || <any expL> is always *true*

Because the result of the comparison is already known, there is no need to evaluate <any expL>. If <any expL> contains a function or method call, it is not called; therefore any side effects of calling that function or method do not occur.

The unary ! operator (logical NOT) returns the opposite of its operand expression. If the expression evaluates to *true*, then !*exp* returns *false*. If the expression evaluates to *false*, !*exp* returns *true*.

Comparison operators

Related topics Example

Comparison operators compare two expressions of the same data type. The comparison returns a logical *true* or *false* value. Comparison operators may be used with character and numeric expressions. Comparing logical expressions is allowed, but redundant; use logical operators instead.

Object references may be compared. For Date objects, the date/time they represent are compared; they may be earlier, later, or exactly the same. For all other objects, only the equality (== and !=) test makes sense. It tests whether two object references refer to the same object.

These are the comparison operators:

Operator	Description
<	Less than
>	Greater than
==	Exactly equal to
!=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

Comparison operators, examples

To see these examples in action, type them into the Script Pad, but don't type the comments.

```
// The usual numeric and string comparisons
? 3 < 4                  // true
? "cat" > "dog"          // false
// Logical comparisons are redundant
valid = true
? valid == true          // true, but so is
? valid                  //   this
? valid == false         // false, but it's simpler to
? !valid                 //   use the logical NOT operator
// Date objects compare the date/time they represent
x = new Date()
y = new Date()           // Should be a few seconds later
? x < y                  // true, date/time in x is before y
x = new Date( "25 Sep 1996" )
y = new Date( "25 Sep 1996" )
? x == y                 // true: objects are different, but date/time is the
same
// Other objects test for equality only
a = new Form()
b = new Form()
c = b
? a == b                 // false, different objects
? b == c                 // true, references to same object
```

Object operators

Related topics

Object operators are used to create and reference objects, properties, and methods. Here are the Object operators:

Operator	Description
new	Creates a new instance of an object
[ ]	Index operator, which accesses the contents of an object through a numeric or string value
. (period)	Dot operator, which accesses the contents of an object through an identifier name
::	Scope resolution operator, to reference a method in a class or call a method from a class.

new operator

The *new* operator creates an object or instance of a specified class.

The following is the syntax for the *new* operator:

```
[<object reference> =] new <class name>([<parameters>])
```

The <object reference> is a variable or property in which you want to store a reference to the newly created object.

Note that the reference is optional syntactically; you may create an object without storing a reference to it. This results in the object being destroyed after the statement that created it is finished, since there are no references to it.

The following example shows how to use the *new* operator to create a Form object from the Form class. A reference to the object is assigned to the variable customerForm:

```
customerForm = new Form();
```

This example creates and immediately uses a StringEx object. The object is discarded after the statement is complete:

```
? new StringEx().replicate( "*", 10 )
```

Index operator

The index operator, [ ], accesses an object's properties or methods through a value, which is either a number or a character string. The following shows the syntax for using the index operator (often called the array index operator):

```
<object reference>[<exp>]
```

You typically use the index operator to reference elements of array objects, as shown in the following example:

```
aScores = new Array(20);  // Create a new array object with 20 elements
aScores[0] = 10;          // Change the value of the 1st element to 10
? aScores[0];             // Displays 10 in results pane of Script Pad
```

Dot operator

The dot operator, (“.”), accesses an object's properties, events, or methods through a name. The following shows the syntax for using the dot operator:

```
<object reference>[.<object reference> ...].<property name>
```

Objects may be nested: the property of an object may contain a reference to another object, and so on. Therefore, a single property reference may include many dots.

The following statements demonstrate how you use the dot operator to assign values:

```
custForm = new Form();          // Create a new form object
custForm.Title = "Customers";   // Set the title property of custForm
```

```
custForm.Color = "Maroon";     // Set the color property of custForm
```

If an object contains another object, you can access the child object's properties by building a path of object references leading to the property, as the following statements illustrate:

```
custForm.addButton = new Button(custForm);  // Create a button in the
custForm form
custForm.addButton.text = "Add";            // Set the text property of
addButton
```

Scope resolution operator

The scope resolution operator (::, two colons, no space between them) lets you reference methods directly from a class or call a method from a class.

The scope resolution operator uses the following syntax:

```
<class name>|class|super::<method name>
```

The operator must be preceded by either an explicit class name, the keyword *class* or the keyword *super*. *class* and *super* may be used only inside a class definition. *class* refers to the class being defined and *super* refers to the base class of the current class, if any.

<*method name*> is the method to be referenced or called.

Scope resolution searches for the named method, starting at the specified class and back through the class's ancestry. Because *super* starts searching in a class's base class, it is used primarily when overriding methods.

There are no formal classes, and therefore no scope resolution operator in client-side JavaScript.

Non-operational symbols

Related topics

Though they don’t act upon data or hold values in themselves, non-operational symbols have their own purpose in IntraBuilder code and in the interpretation of JavaScript programs.The symbols are:

Symbols	Name/meaning
;	Statement terminator
//	End-of-line comment
/* */	Block comment
?	Script Pad results (writeln)
{} {;} {||}	Program/literal array/codeblock markers
"" "	Literal strings
\	String escape character
#	Preprocessor directive

String symbols

Related topics

Quotation marks and double-quotation marks enclose literal strings. The following example simply assigns the string “literal text” to the variable *xString*:

```
xString = "literal text";
```

Numeric values can be merged into a string using the + concatenation operator. However, the number need not be enclosed in quotation marks. Both of these examples assign the string “literal text–4” to the variable *xString*:

```
xString = "literal text" + -4;
xString = 'literal text' + "-4";
```

To specify a quotation mark of the same type as the enclosing marks, you must provide an escape character in front of the internal marks:

```
xString = "literal \"this bit needs quotation marks\" text" + -4;
? xString;
```

The above code returns the following:

```
literal "this bit needs quotation marks" text-4
```

If, however, the needed internal quotation marks are different from the outer enclosing marks, escape characters aren’t necessary. The following code returns the same result as the previous statement:

```
xString = 'literal "this bit needs quotation marks" text' + "-4";
? xString;
```

Comment symbols

Related topics

Two forward slashes (//, no space between them) indicate that all text following the slashes (until the next carriage return) is a comment. Comments let you provide reference information and notes describing your code:

```
x = 4 * y      // multiply the value of y by four and assign the result to
variable x
```

A pair of single forward slashes with “inside” asterisks (/* */) encloses a block comment that can be used for a multi-line comment block:

```
/* this is the first line of a comment block
this is more of the comment
this is the last line of the comment block */
```

You can also use the pair for a comment in the middle of a statement:

```
x = 1000000 /* a million! */ * y
```

Comment blocks cannot be nested. This example shows improper usage:

```
/* this is the first line of a comment block
this is more of the the same /* this nested comment will cause problems*/
this is the last line of the comment block */
```

After the opening block marker, IntraBuilder ends the comment at the next closing block marker it finds, which means that only the section of the comment from “this is the first line” to the word “problems” will be interpreted as a comment. The unenclosed remainder of the block will generate an error.

Statement terminator

Related topics

The semicolon (;) is a statement terminator.

In theory, you never need to use a statement terminator in IntraBuilder code. Although IntraBuilder compiler always recognizes the semicolon as a terminator, it will also compile most properly constructed code that contains
no terminator. If, for example, you type this into the Script Pad:

```
a = 2 b = 5 * a
? b
```

the answer, 10, is printed in the results pane, just as if you had typed this:

```
a = 2;
b = 5 * a;
? b
```

In practice, however, statement terminators promote readability and make debugging easier. This example, typed in the Script Pad, displays the result “Games not held” in the results pane:

```
c = 1900 x = 19 o = new Object() o[ 2000 ] = "Sydney" o[ 1996 ] = "Atlanta"
o[ 1977 ] = "Games not held" o[1984] = "Los Angeles" y = new Date().getYear()
_sys.scriptOut.writeln( o[ c + y - x] )
```

But the same code segment, with terminators and comments added, would be much easier to read and debug if you need to review or update the code later:

```
c = 1900;
/* getYear() returns only a two-digit value, so we need to add the century;
change this to 2000 when the time comes */
x = 19;                                  // test value
o = new Object();                        // create the array object
o[ 2000 ] = "Sydney";                    // add values to the array
o[ 1996 ] = "Atlanta";
o[ 1977 ] = "Games not held";
o[ 1984 ] = "Los Angeles";
y = new Date().getYear();                // get the current year (1996)
_sys.scriptOut.writeln( o[ c + y - x] );   // returns "Games not held"
```

For testing small segments of code in the Script Pad, it’s sometimes more efficient to simply enter your statements without terminators, using carriage returns instead or simply typing a stream of statements. The first example above, for instance, can be run successfully by simply pasting directly into the Script Pad from your online Help file, whereas you would have to remove the comment lines from the second version before pasting it into the pad to test the result.

Script Pad results symbol

Related topics

The question mark (?) instructs IntraBuilder to display the result of one or more expressions in the Script Pad results pane. It is a shortcut for _sys.scriptOut.writeln() intended to be used in the Script Pad only; never in scripts.

The results symbol must be followed by one or more valid evaluation expressions, separated by commas. If nothing follows the results symbol, an error message (“Expression expected”) appears.

The following are examples of valid evaluation expressions, with the results pane display shown below the line:

```
? 2 * 4
x = 10 - 5
? "Now is the time..."
? This generates an error message, because "This" is viewed as an unassigned
variable
? x
? 9
================================================================================
=======
8
Now is the time...
5
9
```

String escape symbol

Related topics

The backslash (\) is used inside string expressions to inform IntraBuilder that the next character is a character and not another symbol. It's only needed when you want to include characters that would normally be translated as symbols, such as quotation marks, apostrophes and backslashes. For example, use

```
cBlurb = "The product was named \"Product of the Year\" by the Marketing\\
Sales Association."
```

to get this result:

```
The product was named "Product of the Year" by the Marketing\Sales
Association.
```

Each instance of characters that require it must be preceded (with no space) by the escape symbol, or IntraBuilder will perform literal translation. For example,

```
cBlurb = "The product was named \""Most Improved" Product of the Year\" by
the Marketing\\Sales Association."
```

results in an error because IntraBuilder considers the cBlurb string terminated at "The product was named \"", then regards Most as an undefined variable. To make the string work as a whole, precede all of the "inner" quotation marks with escape characters:

```
cBlurb = "The product was named \"\"Most Improved\" Product of the Year\" by
the Marketing\\Sales Association."
```

This example results in the following string:

```
The product was named ""Most Improved" Product of the Year" by the Marketing\
Sales Association.
```

Escape symbols are required for quotation marks only if the marks are the same as those that enclose the string. Thus, if you use double quotation marks to enclose the string, as above, you wouldn't have to use the escape symbol on any single quotation marks within the string. For example,

```
cBlurb = "The product was named \"'Most Improved' Product of the Year\" by
the Marketing\\Sales Association."
```

results in:

```
The product was named "'Most Improved' Product of the Year" by the Marketing\
Sales Association.
```

Conversely,

```
cBlurb = 'The product was named "Most Improved Product of the Year" by the
Marketing\\Sales Association.'
```

results in:

```
The product was named "Most Improved Product of the Year" by the Marketing\
Sales Association.
```

You may also insert some special characters in a literal string by using one of the following backslash-and-character combinations:

Combination	Result	ASCII value
\b	backspace	8
\f	form feed	12
\n	new line	10
\r	carriage return	13
\t	tab	9

One final note on the use of the backslash escape symbol: If it is not followed by a character that requires it, IntraBuilder simply ignores the backslash. For example, "S" does not require the escape symbol, so

```
cBlurb = 'the Marketing\Sales Association'
```

results in:

```
the MarketingSales Association
```

{};{;} Statement block, codeblock, array symbol

Related topics

Braces ({ }) enclose statement blocks, codeblocks, and literal array elements. They must always be paired. The following examples show how braces may be used in IntraBuilder code.

To enclose statement blocks

```
class test2Form extends Form {
   with (this) {
      height = 20;
      left = 65;
      top = 0;
      width = 60;
      title = "";
   }
}
```

To enclose arrays

```
a = {1,2,3};
a[1]           // returns "2" (the second item in the zero-based array)
```

To assign a statement codeblock to an object's event handling property

```
form.onLoad = {;alert("Warning: You are about to enter a restricted area.")}
```

To assign an expression codeblock to a variable, and pass parameters to it

```
c = {|x| x*9};
? c(4)             // returns 36
// or
q = {|n| {"1st","2nd","3rd"} [n-1]};
? q(2)                                    // returns "2nd"
```

To assign an expression codeblock to a variable, without passing parameters

```
c = {|| 4*9};      // pipes (||) must be included in an expression codeblock,
                   // even if a parameter is not being passed
? c()              // returns 36
```

Preprocessor directive symbol

Related topics

The number sign (#) marks preprocessor directives, which provide instructions to the IntraBuilder compiler. Preprocessor directives may be used in scripts only.

Use directives in your IntraBuilder code to perform such compile-time actions as replacing text throughout your program, perform conditional compilations, include other source files, or specify compiler options.

The symbol must be the non-blank first character on a line, followed by the directive (with no space), followed by any conditions or parameters for the directive.

For example, you might use this statement:

```
#include "IDENT.H"
```

to include a source file named IDENT.H (the “H” extension us generally used to identify the file as a “header” file) in the compilation. The included file might contain its own directives, such as constant definitions:

```
//file IDENT.H: constant definitions for MYPROG
#define COMPANY_NAME  "Nobody's Business"
#define NUM_EMPLOYEES 1
#define COUNTRY       "Liechtenstein"
```

For a complete listing of all IntraBuilder preprocessor directives, along with syntax and examples for each, see Preprocessor.

Other symbols and symbol usage notes

Related topics

IntraBuilder code neither requires nor recognizes a statement continuation symbol (such as the “\” used in C++ code).

Core language

The programming language used in IntraBuilder is an enhanced version of JavaScript. Among its additional features are

- Formal *class* declaration with single inheritance, making IntraBuilder a fully object-oriented programming platform
- Exception handling with *try*, *catch*, *finally*, and *throw*

In addition to the reserved words that comprise the core of the JavaScript language, this section of the *Language Reference* also documents

- The built-in JavaScript functions *eval*(), *parseFloat*(), *parseInt*(), *escape*(), and *unescape*().
- The Exception class used in exception handling, and the generic Object class
- The *className* property, *parent* property, and *release*() method, common to most objects in IntraBuilder

class Exception

Related topics Example

An object that describes an exception condition.

Syntax

[<oRef> =] new Exception()

<oRef>

A variable or property in which to store a reference to the newly created Exception object.

Properties

The following table lists the properties of the Exception class. (No events or methods are associated with this class.)

Property	Default	Description
className	Exception	Identifies the object as an instance of the Exception class
code		A numeric code to identify the type of exception
message	Empty string	Text to describe the exception

Description

An Exception object is automatically generated by IntraBuilder whenever an error occurs. The object's properties contain information about the error.

You can also create an Exception object manually, which you can fill with information and *throw* to manage execution or to jump out of deeply nested statements.

You may subclass the Exception class to create your own custom exception objects. A *try* block may be followed by multiple *catch* blocks, each one looking for a different exception class.

class Exception example

Suppose you are using exceptions to manage execution in a deeply nested set of conditional statements and loops. You create your own exception class:

```
class JumpException extends Exception
{
}
```

Then in the code, you create the JumpException object and *throw* it if needed:

```
try {
  var j = new JumpException();
  // Lots of nested code
  Ä
                        if ( lItsNoGood ) {
                            throw j; // Deep in the code, you want out
                        }
  Ä
}
catch ( JumpException e )
  // Do nothing; JumpException is OK
}
catch ( Exception e )
  // Normal error
  logError( new Date(), e.message ); // Record error message
  // and continue
}
```

If there is a normal error, the second *catch* block saves it to a log file, using a function you wrote, and execution continues.

class Object

Related topics Example

An empty object.

Syntax

[<oRef> =] new Object()

<oRef>

A variable or property in which to store a reference to the newly created object.

Properties

An object of the Object class has no initial properties, events, or methods.

Description

Use the Object class to create your own simple objects. Once the new object is created, you may add properties and methods through assignment. You cannot add events.

This technique of adding properties and methods on-the-fly is known as dynamic subclassing. In IntraBuilder, dynamic subclassing supplements formal subclassing, which is achieved through *class* definitions.

The Object class is the only class in IntraBuilder that does not have the read-only *className* property.

class Object example

The following statements create a simple object with a few properties—some referenced by name and some referenced by number—and a codeblock as a method.

```
o = new Object();
o.title = "Summer";
o[ 2000 ] = "Sydney";
o[ 1996 ] = "Atlanta";
o.cityInYear = {|y| this[ y ]};
_sys.scriptOut.writeln( o.cityInYear( 2000 ) ); // Displays "Sydney"
```

break

Related topics Example

Immediately terminates the current *for* or *while* loop. Execution continues with the statement after the loop.

Syntax

break

Description

Normally, a *for* loop executes a certain number of times, and a *while* loop executes while the specified condition is *true*. All of the statements in the loop are executed in each iteration of the loop; in other words, the loop always exits after the last statement in the loop.

Use *break* to exit a loop from the middle of a loop, due to some extra or abnormal condition. In most cases, you don't have to resort to using a *break*; you can code the *for* or *while* condition, which controls the loop to handle the extra condition. The condition is tested between loop iterations, after the last statement, but that usually means that there are some statements that should not be executed because of this condition. Those statements would have to be conditionalized out with an *if* statement. Therefore, often it's simpler to *break* out of a loop immediately once the condition occurs.

break example

The following function counts the number of words in a string by counting spaces between words. Multiple spaces between two words are counted as a single space and therefore a single word:

```
function wordCount( cArg ) {
  var nRet    = 0;                     // Initialize counter
  var cRemain = new StringEx( cArg );
  cRemain.rightTrim();                 // Remove spaces around text
  cRemain.leftTrim();
  while ( cRemain.length > 0 ) {       // Something left to check
    nRet++;                            // Increment word count
    var nPos = cRemain.indexOf( " " ); // Find next space
    if ( nPos == -1 )                  // No more spaces
      break;                           //   means no more words
    cRemain.string = cRemain.substring( nPos, cRemain.length );
    cRemain.leftTrim();                // Otherwise remove first word
                                       //   and continue
  }
  return nRet;
}
```

The condition in the *while* loop is really needed only once, the first time the loop is entered. It makes sure that there is some text to search through. If the argument is an empty string or all spaces, the loop is not executed and the word count is zero. After the first loop, it is used simply to keep the loop going, since there would always be text to check.

The loop is terminated when there are no more spaces in the string. This is determined by the return value of the *indexOf*() method. Because the index returned is out of range for the *substring*() method, it should be called if there are no more spaces in the string. By using *break*, the loop is immediately terminated once no more spaces are found. Execution continues with the *return* statement following the *while* loop.

case

Designates a block of code in a *switch* block.

Description

See *switch* for details.

catch

Designates a block of code to execute if an exception occurs inside a *try* block.

Description

See *try* for details.

class

Example

A class declaration including constructor code, which typically creates member properties, and class methods.

Syntax

class <class name> [*extends* <superclass name> [*custom*]]{
    [<constructor code>]
    [<methods>]
}

<class name>

The name of the class.

extends <superclass name>

Indicates that the class is a derived class which inherits the methods defined in the superclass. The superclass constructor is called before the <constructor code> in the *class* is called, which means that any properties created in the superclass are inherited by the class.

custom

Identifies the class as a custom component class, so that its predefined properties are not streamed out by the Form or Report Designer.

<constructor code>

The code that is called when a new instance of the class is created with the *new* operator. The constructor consists of all the code at the top of the class declaration up to the first method.

<methods>

Any number of functions designed for the class.

Description

A *class* declaration formalizes the creation of an object and its methods. Although you can always add properties to an object and assign methods dynamically, a *class* simplifies the task and allows you to build a clear class hierarchy.

Another benefit is polymorphism. Every *function* defined in the *class* becomes a method of the class. An object of that class automatically has a property with the same name as each *function* and which contains a reference to that *function*. Because a method is part of the *class*, different functions may use the same name as long as they are methods of different classes. For example, you can have multiple copy() functions in different classes, with each one applying to objects of that class. Without classes, you would have to name the functions differently even if they performed the same task conceptually.

Before the first statement in the constructor is executed, if the *class* extends another class, the constructor for that superclass has already been executed, so the object contains all the superclass properties. Any properties that refer to methods, as described in the previous paragraph, are assigned. This means that if the *class* contains a method with the same name as a method in a superclass, the method in the *class* overrides the method in the superclass. The *class* constructor, if any, then executes.

In the constructor, the reserved word *this* refers to the object being created. Typically, the constructor creates properties by assigning them to *this* with dot notation. However, the constructor may contain any code at all, except another *class*—you can't nest classes—or a *function*, since that *function* would become a method of the class and indicate the end of the constructor.

class example

The following class extends the Form class and changes the *color* property from silver to lemonchiffon:

```
class MyForm extends Form
{
  this.color = "lemonchiffon";
}
```

className

Related topics

Identifies of which class the object is a member.

Property of

All classes except Object.

Description

The *className* property identifies the class constructor that originally created the object. Although you may dynamically subclass the object by adding new properties, the *className* property does not change.

The *className* property is read-only.

continue

Related topics

Skips the remaining statements in the current *for* or *while* loop, causing another loop iteration to be attempted.

Syntax

continue

Description

Conditional statements are often used inside a loop to control which statements are executed in each loop iteration. For example, in a loop that processes the rows in an employee table, you might want to increase the monthly salary of non-managers and the annual bonus for managers, all in the same loop.

There can be many different sets of statements in the loop, each with a different combination of conditions dictating whether they should be executed. Sometimes you can be in the middle of a loop, and none of the remaining statements apply. The condition that determines this may be nested a few levels deep. While it would be possible to code the rest of the loop with conditional statements to take this condition into account, often it's simpler to use a *continue* statement when this condition is encountered. This causes the remaining statements in the loop to be skipped, and another iteration of the loop to be attempted.

default

Designates a block of code in a *switch* block to execute if there are no matching *case* blocks.

Description

See *switch* for details.

else

Designates an alternate statement to execute if the condition in an *if* statement is *false*.

Description

See *if* for details.

escape()

Related topics Example

Encodes a string, replacing characters with their ASCII equivalents.

Syntax

escape(<expC>[, <bitmask expN>])

<expC>

The character expression to encode.

<bitmask expN>

Controls how the string is encoded. The individual bits of the numeric expression <bitmask expN> are evaulated in the order they are listed in the folowing table.

Bit	Decimal value	Option
bit 2	4	Encode unsafe characters except “/” and “+”
bit 0	1	Encode all unsafe characters
bit 1	2	Encode unsafe characters and encode space as “+”
All bits zero	0	Encode all characters

If the bit corresponding to an option is on—that is, set to 1—then that option is used. The options are mutually exclusive. For example, if <bitmask expN> is 3 (both bit 0 and bit 1 on), it is considered to be 1 (bit 0).

The default value of <bitmask expN> is 4: encode unsafe characters except “/” and “+”.

Description

Use *escape*() to encode a character string to replace unsafe characters. Most non-alphanumeric characters, like spaces and most punctuation marks, are considered unsafe; they cannot be used in URLs. By using *escape*() to encode those characters, the value of the string can be passed in a URL in its encoded form. Then the *unescape*() function can be used to decode the string to get its actual value.

The encoding for a character is its ASCII value in the ISO Latin-1 character set in hexadecimal, preceded by the “%” character. For example,

```
escape( "hi!" )    // returns "hi%21"
escape( "a,b" )    // returns "a%2Cb"
```

Although there is an option to encode spaces as plus signs, the *unescape*() function will never decode a plus sign as a space (it is always considered to be a plus sign). Therefore if that encoding option is used, you must manually replace plus signs in the encoded string with spaces before using the *unescape*() function.

escape() example

The following custom Header for the report SEARCH.JRP takes the page to display as the first parameter and an SQL statement as the second parameter. The link to display the next page in the report, which is managed by the report's *isLastPage*() method, must contain the SQL statement. Because the SQL statement contains spaces, it cannot be used as-is in the URL; therefore it is encoded with the *escape*() function before it is stored.

When the report is called with the encoded SQL statement, the *unescape*() function is used to decode the string. To determine whether the SQL statement is encoded, the *indexOf*() method is used; if the string contains a space, it is not encoded.

```
var r = new SEARCHReport();
if (SEARCH.arguments.length == 2) {
  var sql = SEARCH.arguments[1];
  if ( sql.indexOf( " " ) >= 2) {
    // If there's a space, it's a plain SQL statement
    r.query1.sql = sql;
    r.encodedSQL = escape( sql );
  }
  else {
    // otherwise, it's encoded
    r.query1.sql = unescape( sql );
    r.encodedSQL = sql;
  }
}
else {
  r.encodedSQL = "";
}
if (SEARCH.arguments.length >= 1) {
   r.startPage = r.endPage = SEARCH.arguments[0];
}
r.render();
return;
```

eval()

Related topics Example

Evaluates the contents of a string as a JavaScript expression.

Syntax

eval(<expC>)

<expC>

The expression to evaluate.

Description

eval() acts like a miniature JavaScript interpreter that evaluates JavaScript code at run time. It evaluates the contents of a string, as if that string was a JavaScript expression, and returns the value of that expression.

When the string contains a number, *eval*() acts like *parseFloat*() and *parseInt*(), except that *eval*() does not simply stop at the first non-numeric character as *parseFloat*() and *parseInt*() do.

eval() treats the entire string as an expression, so the string may contain any expression elements, including literals, operators, variables, properties, and function calls. The expression described in the string may be of any type, just like an expression typed into a script or the Script Pad.

eval() example

The following examples illustrate *eval*() and how it differs from *parseFloat*() and *parseInt*():

```
? parseInt( "45" )            // Displays 45
? eval( "45" )                // Displays 45
? parseInt( "3 + 2" )         // Displays 3
? eval( "3 + 2" )             // Displays 5
? parseFloat( "212 Baker" )   // Displays 212.00
? eval( "212 Baker" )         // Error, invalid expression
                              // (exact error depends on version of
JavaScript)
? parseFloat( "true" )        // Displays 0
? eval( "true" )              // Displays true
```

The following statement reads some code stored as a string in a character field in a table and uses *eval*() to assign that code in a codeblock as the event handler for another field in another table.

```
this.form.rowset.fields[ "Shares sold" ].canChange = eval( "{|newValue|" +
  this.form.qRules.rowset.fields[ "Sell rules" ].value + "}" );
```

This is an example of a data-driven application, where code is stored externally in tables. It allows the application to be changed without touching the scripts.

finally

Designates a block of code that always executes after a *try* block, even if an exception occurs.

Description

See *try* for details.

for

Related topics Example

A control statement designed to execute a statement or block of statements a certain number of times.

Syntax

for ([<initial exp> [, <exp> [, ...]]] ; [<condition expL>] ; [<update exp> [, <exp> [, ...]]])
    <statement> | { <statement block> }

<initial exp>

An optional expression that is evaluated once before the *for* loop begins. It is usually used to set the initial value of the loop counter, a variable or property that maintains a count of the number of times the loop has been executed and is often used inside the loop. You may specify multiple expressions by separating them with commas.

<condition expL>

An optional logical expression that is evaluated before each iteration of the loop to determine whether the iteration should occur. The expression usually compares the loop counter set in <initial exp> against some other value. If no expression is specified, it's assumed to always be *true*, resulting in an infinite loop.

<update exp>

An optional expression that is evaluated after each iteration of the loop. The expression usually increments or decrements the loop counter set in <initial exp>. You may specify multiple expressions by separating them with commas.

<statement> | { <statement block> }

A single statement or a statement block inside curly braces that is executed in each iteration of the loop.

Description

Use a *for* loop to execute a statement or block of statements a certain number of times. The *for* loop formalizes the structure and syntax for a counted loop; you can create counted loops using *while*.

for loop example

The following statements demonstrate two nested loops that traverse all the columns in all the rows of a two-dimensional array and display their contents in the results pane of the Script Pad. The variable nRows contains the number of rows in the array, and the variable nCols contains the number of columns. nColWidth is a variable that contains the width of each column you want to display, and aArg is the array in question. Array rows and columns are numbered from zero.

```
for ( var nRow = 0; nRow < nRows; nRow++ ) {
  _sys.scriptOut.writeln();                    // Each row on its own line
  for ( var nCol = 0; nCol < nCols; nCol++ ) {  // Display each row as before
    _sys.scriptOut.column = nColWidth * nCol;
    _sys.scriptOut.write( aArg[ nRow, nCol ] );
  }
}
```

for...in

Related topics Example

A control statement that loops through all the properties of an object.

Syntax

for (<var> *in* <oRef>)
    <statement> | { <statement block> }

<var>

A variable that is assigned the name of a property of the object in each iteration of the loop.

<oRef>

A reference to the object whose properties you want to loop through.

<statement> | { <statement block> }

A single statement or a statement block inside curly braces that is executed in each iteration of the loop.

Description

Use *for...in* to loop through all the properties in an object. The variable <var> is assigned the name of each property, as a string, in each iteration of the loop. Get the value of the property by concatenating the property name with the object reference and evaluating that string with the built-in *eval*() function.

for...in example

The following statements display the contents of a new Form object:

```
var f = new Form()
for ( x in f ) {
  _sys.scriptOut.writeln( x );
  _sys.scriptOut.column = 15;
  _sys.scriptOut.write( eval( "f." + x ) );
}
```

The following function returns *true* or *false* to indicate whether the given name is a property of the object:

```
function isProperty( obj, cPropName )
{
  for ( var c in obj ) {
    if ( c == cPropName ) {
      return true;
    }
  }
  return false;
}
```

function

Related topics Example

Defines a function in a script including variables to represent parameters passed to the function.

Syntax

function <function name>([<parameter list>])
  { [<statements>] }

<**function** name**>**

The name of the function. Although IntraBuilder imposes no limit to the length of function names, it recognizes only the first 32 characters.

(<parameter list>)

Variable names to assign to data items (or parameters) passed to the function by the statement that called it.

<statements>

Any valid statements that you want the function to execute. You can call functions recursively.

Description

Use functions to create code modules. By putting commonly used code in a function, you can easily call it whenever needed, and pass parameters to the function.

When a *function* is defined inside a *class* definition, the *function* is considered a method of that *class*.

When a *function* is called via an object, usually as a method or event handler, the reserved word *this* refers to the object that called the function.

function example

The function isBlank() receives the character string parameter cArg and returns a logical value to indicate whether the string is blank.

```
_sys.scriptOut.writeln( isBlank( "abc" ) );  // Displays false
_sys.scriptOut.writeln( isBlank( "   " ) );  // Displays true
function isBlank( cArg ) {
  var c = cArg;                       // Make a work copy
  while ( c.length > 0 && c.charAt( 0 ) == " " ) {
    c = c.substring( 1, c.length );  // Remove first character (a space)
  }
  return c == "";
}
```

if

Example

Conditionally executes a statement or block of statements.

Syntax

if (<condition expL>)
    <statement> | { <statement block> }
[*else*
    <statement> | { <statement block> }]

<condition expL>

A logical expression that determines if the statement or block of statements after the *if* execute. If the condition is *true*, the statements execute.

<statement> | { <statement block> }

A single statement or a statement block inside curly braces that execute depending on the value of <condition expL>.

else <statement> | { <statement block> }

Specifies a single statement or a statement block inside curly braces to execute if <condition expL> is *false*.

Description

Use *if* to execute one set of statements or another, depending on the value of a logical condition. If the condition is based on the value of a numeric expression, using *switch* may be more efficient and flexible. If you're evaluating a condition to decide which value you want to assign to a variable or property, you may be able to use the ?: conditional operator, which involves less duplication (you don't have to type the target of the assignment twice).

If the condition is not true, the statements in the *else* block, if there is one, are executed. You may include another *if* inside the *else* and repeat the process to evaluate a number of conditions in a row.

if example

The following *onServerSubmit* event handler for a form runs a different form or report, depending on the value in a Hidden object on the form. If the value happens to be “VIEWER”, then it also passes the value of another Hidden object, if that value is greater than zero.

```
function Form_onServerSubmit()
{
  if ( this.hiddenAction.value == "VIEWER" ) {
    if ( parseInt( this.hiddenMsg.value ) > 0 ) {
      _sys.forms.run( "VIEWER", parseInt( this.hiddenMsg.value ) );
    }
    else {
      _sys.forms.run( "VIEWER" );
    }
  }
  else if ( this.hiddenAction.value == "NEW" ) {
    _sys.reports.run( "NEWMSG" );
  }
  else if ( this.hiddenAction.value == "ADMIN" ) {
    _sys.forms.run( "ADMIN" )
  }
}
```

parent

Related topics

The immediate container of an object.

Property of

Most data access, form, and report objects

Description

Many objects are related in a containership hierarchy. If the container object— referred to as the parent —is destroyed, all the objects it contains—referred to as child objects—are also destroyed. Child objects may be parents themselves and contain other objects. Destroying the highest-level parent destroys all the descendant child objects.

An object's *parent* property refers to its parent object.

For example, a form contains both data access objects and visual components. A Query object in a form has the form as its parent. The Query object contains a rowset, which contains an array of fields, which in turn contains Field objects. Each object in the hierarchy has a *parent* property that refers back up the chain, up to the form, which has no parent. A button on the form also has a *parent* property that refers to the form. If the form is destroyed, all of the objects it contains are destroyed.

The *parent* property is often used to refer to sibling objects—other objects that are contained by the parent. For example, one Field object can refer to another by using the *parent* reference to go one level up in the hierarchy, then use the name of the other field to go back down one level to the sibling object.

The *parent* property is read-only.

parent example

parseFloat()

Related topics Example

Converts an expression to a floating point number.

Syntax

parseFloat(<exp>)

<exp>

The expression you want to convert.

Description

parseFloat() returns the floating point number equivalent of an expression, using the following rules:

- Integers are returned with the same value, but IntraBuilder displays them as floating point numbers.
- Floating point numbers are returned unchanged.
- Strings that start with a digit or sign (+ or –) are converted to floating point numbers, up to the first character that is not a digit, a decimal point, or the letter E (or e) designating an exponent.
- Everything else is converted to 0 or NaN (not a number), depending on the version of JavaScript. IntraBuilder does not support NaN.

parseFloat() example

The following examples illustrate the *parseFloat*() conversion rules:

```
? parseFloat( 4 )              // Displays 4.00
? parseFloat( 4.56 )           // Displays 4.56
? parseFloat( -3.2 )           // Displays -3.2
? parseFloat( "45" )           // Displays 45.00
? parseFloat( "212 Baker" )    // Displays 212.00
? parseFloat( "-3e2" )         // Displays -300.00
? parseFloat( null )           // Displays 0.00
```

parseInt()

Related topics Example

Converts an expression to an integer.

Syntax

parseInt(<exp>[, <radix expN>])

<exp>

The expression you want to convert.

<radix expN>

The numeric base for interpreting the number. The valid range is 2 to 36. The default is 10.

Description

parseInt() returns the integer equivalent of an expression, using the following rules:

- Integers are returned unchanged.
- Floating point numbers are truncated to integers.
- Strings that start with a digit or sign (+ or –) are converted to integers, up to the first non-digit character.
- Everything else is converted to 0 or NaN (not a number), depending on the version of JavaScript. IntraBuilder does not support NaN.

While *parseInt*() defaults to interpreting the number or string as a decimal number, it can convert numbers with another radix, or base. For example, if <radix expN> is 8, the number is considered an octal number; if it is 16, the number is hexadecimal.

For bases higher than 10, the letters A through Z (case-insensitive) are considered digits representing the numbers 10 through 35. For bases less than 10, the digits greater than or equal to the radix are considered to be non-digit characters, which stop the conversion.

parseInt() example

The following examples illustrate the *parseInt*() conversion rules:

```
? parseInt( 4 )             // Displays 4
? parseInt( 4.56 )          // Displays 4
? parseInt( -3.2 )          // Displays -3
? parseInt( "45" )          // Displays 45
? parseInt( "212 Baker" )   // Displays 212
? parseInt( "-3e2" )        // Displays -3
? parseInt( null )          // Displays 0
```

These examples show different radixes:

```
? parseInt( "31" )          // Displays 31
? parseInt( "31", 8 )       // Displays 25
? parseInt( "64", 16 )      // Displays 100
? parseInt( "we", 36 )      // Displays 1166 (= 32 * 36 + 14)
? parseInt( "123", 3 )      // Displays 5 (= 1 * 3 + 2; 3 is non-digit in
base 3)
```

quit

Example

Closes all open files and terminates IntraBuilder.

Syntax

quit

Description

Use *quit* to end your IntraBuilder work. It has the same effect as closing the IntraBuilder application.

If you include *quit* in a script, IntraBuilder halts the script's execution and exits IntraBuilder. *quit* is usually not used in scripts, forms, or reports that are executed remotely, since it would terminate the IntraBuilder Agent, closing all connections to that IntraBuilder Agent, including those with other users.

quit example

At the end of a long day, suppose you want to exit IntraBuilder and run the latest 3-D shoot-em-up game, which requires 128 MB of RAM. Your hands are already on the home keys of the keyboard, so instead of reaching to press Alt-F4 or using the mouse to click the close button, you type the following in the Script Pad.

```
quit
```

release()

Explicitly releases an object from memory.

Syntax

<oRef>.*release*()

<oRef>

An object reference to the object you want to release.

Property of

All form objects: ActiveX, Button, CheckBox, Form, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea; all report objects except Band and StreamFrame: Group, PageTemplate, Report, StreamSource.

Description

IntraBuilder manages memory and resources used by objects automatically. When there are no more variables or properties that reference an object and that object is not visible onscreen, the object is destroyed. Any components that are contained in the object, such as the components of a form, are also destroyed when the container is destroyed. Because of this automatic object management, there is usually no reason to call *release*().

release() explicitly releases an object from memory. Any references that point to that object become invalid; attempting to use such a reference results in an error.

For example, you might want to get rid of a single component in a form. You could *release*() that component, but in most cases you could just as easily hide the component by setting its *visible* property to *false*.

return

Related topics Example

Ends execution of a script or function, returning control to the calling routine—script or function—or to the Script Pad.

Syntax

return [<return exp>]

<return exp>

The value a function returns to the calling routine or the Script Pad.

Description

Scripts and functions return to their callers when there are no more statements to execute. When ended this way, they do not return a value.

Use *return* in a script or function to return a value, or to return before the end of the script or function.

If the *return* is inside a *try* block, the corresponding *finally* block, if any, is executed before returning. If there is a *return* inside that *finally* block, whatever it returns is returned instead.

return example

The function isBlank() tests a character string and returns a logical value to indicate whether the string is blank.

```
_sys.scriptOut.writeln( isBlank( "abc" ) );  // Displays false
_sys.scriptOut.writeln( isBlank( "   " ) );  // Displays true
function isBlank( cArg ) {
  var c = cArg;                       // Make a work copy
  while ( c.length > 0 && c.charAt( 0 ) == " " ) {
    c = c.substring( 1, c.length );  // Remove first character (a space)
  }
  return c == "";
}
```

switch

Related topics Example

Conditionally executes statements based on the value of a numeric expression.

Syntax

```
switch ( <expN> ) {
   [case <constant expN1>:
      [<statement block>]
      [break;]
   [case <constant expN2>:
      [<statement block>]
      [break;]
   [case ...]]]
   [default:
      [<statement block>]]
}
```

<expN>

A numeric expression to evaluate.

case <constant expN>

Where execution goes if <expN> equals <constant expN>. <constant expN> should be a non-negative integer constant. Each integer may be used in only one *case* statement in a *switch* block.

<statement block>

Any number of statements that are executed when <expN> equals <constant expN> in the preceding *case* statement.

break

Exits *switch* block and continues execution with the next statement after the *switch* block.

default

If present, executes the <statement block> that follows if the <expN> does not match any <constant expN>.

Description

Use *switch* when testing a numeric expression that might evaluate to a number of known values. Using *switch* is more efficient than using *if* with a compound logical expression to test the same expression, because the expression is evaluated only once.

switch works by diverting execution to the *case* that contains the matching numeric value. Once the jump has been made into a *case*, the remaining *case* and *default* statements are ignored, and the rest of the statements in the *switch* block are executed from the top down. There is usually a *break* at the end of each <statement block> so that only the statements for that case are executed, but you can construct a *switch* block to take advantage of the top-down execution. For example, suppose you have created the following #define preprocessor directives for different service levels in a subscription service:

```
#define SUB_LIFETIME 0
#define SUB_PREMIUM  1
#define SUB_STANDARD 2
#define SUB_TRIAL    3
```

For a particular action, the Lifetime subscription is the same as the Premium, and the Premium gets something extra in addition to the Standard. The Trial gets something different. You could code a *switch* block like this:

```
switch ( this.form.rowset.fields[ "Sub level" ].value ) {  // Numeric value
from table
  case SUB_LIFETIME:   // No statements, same as Premium
  case SUB_PREMIUM:
    // Stuff for Premium and Lifetime
  case SUB_STANDARD:
```

```
    // Stuff for Standard, Premium, and Lifetime
    break;  // Stop!
  case SUB_TRIAL:
    // Stuff for Trial only
}
```

There is no *break* after the Trial statements, because there are no more statements in the *switch* block. A *break* could be placed there; it would have no effect.

The integer constants are used in *case* statements in numeric order, but that is not required. The numbers also do not have to be contiguous, but that is often the case.

A *switch* block must contain at least one *case* or the *default*.

switch example

The following function displays the contents of an Array object in the results pane of the Script Pad. It uses a *switch* statement to handle one- and two-dimensional arrays differently. It does not handle arrays with more than two dimensions, displaying a message instead.

function displayArray(aArg, nColWidth)

```
{
  #define DEFAULT_WIDTH 2
  if ( displayArray.arguments.length < 2 ) {
    nColWidth = DEFAULT_WIDTH;
  }
  var cLine = new StringEx();
  switch ( aArg.dimensions ) {
    case 1:                                                 // 1-D
      _sys.scriptOut.writeln( cLine.replicate( "-", nColWidth * aArg.length )
);
      _sys.scriptOut.writeln();
      for ( var nElement = 0; nElement < aArg.length; nElement++ ) {
        _sys.scriptOut.column = nColWidth * nElement; // Line up columns
        _sys.scriptOut.write( aArg[ nElement ] );     // Display elements
      }                                                 //   in a single line
      break;
    case 2:                                                 // 2-D
      var nCols = aArg.subscript( aArg.length - 1, 2 ) + 1 // Determine # of
columns
      var nRows = aArg.length / nCols;                  // Calculate # of rows
      _sys.scriptOut.writeln( cLine.replicate( "-", nColWidth * nCols ) );
      for ( var nRow = 0; nRow < nRows; nRow++ ) {
        _sys.scriptOut.writeln();                       // Each row on its own
line
        for ( var nCol = 0; nCol < nCols; nCol++ ) {  // Display each row as
before
          _sys.scriptOut.column = nColWidth * nCol;
          _sys.scriptOut.write( aArg[ nRow, nCol ] );
        }
      }
      break;
    default:
      alert( "Error: only 1 or 2 dimensions allowed" );
  }
}
```

throw

Related topics Example

Generates an exception.

Syntax

throw <exception oRef>

<exception oRef>

A reference to the Exception object you want to pass to the *catch* handler.

Description

Use *throw* to manually generate an exception. *throw* must pass a reference to an existing Exception object that describes the exception.

throw example

Suppose you are using exceptions to manage execution in a deeply nested set of conditional statements and loops. You create your own exception class:

```
class JumpException extends Exception
{
}
```

Then, in the code, you create the JumpException object and *throw* it if needed:

```
try {
  var j = new JumpException();
  // Lots of nested code
  Ä
                       if ( lItsNoGood ) {
                           throw j; // Deep in the code, you want out
                       }
  Ä
}
catch ( JumpException e )
  // Do nothing; JumpException is OK
}
catch ( Exception e )
  // Normal error
  logError( new Date(), e.message ); // Record error message
  // and continue
}
```

If there is a normal error, the second *catch* block saves it to a log file, using a function you wrote, and execution continues.

try

Related topics Example

A control statement used to handle exceptions and other deviations of program flow.

Syntax

```
try
    { <statement block 1> }
[ catch( <exception type1> <exception oRef1> )
    { <statement block 2> } ]
[ catch( <exception type2> <exception oRef2> )
    { <statement block 3> } ]
[ catch ... ]
[ finally
    { <statement block 4> } ]
```

try { <statement block 1> }

A statement block inside curly braces for which the following *catch* or *finally* block—or both—will be used if an exception occurs during execution. A *try* block must be followed by either a *catch* block, a *finally* block, or both.

catch { <statement block 2> }

A statement block inside curly braces that is executed when an exception occurs.

<exception type>

The class name of the exception to look for—usually, Exception.

<exception oRef>

A formal parameter to receive the Exception object passed to the *catch* block.

catch...

Catch blocks for other types of exceptions.

finally { <statement block 2> }

A statement block inside curly braces which is always executed after the *try* block, even if an exception or other deviation of program flow occurs. If there is both a *catch* and a *finally*, the *finally* block executes after the *catch* block.

Note There is no semicolon between the closing brace (}) and the keyword *catch* or *finally*.

Description

An exception is a condition that is either generated by IntraBuilder, usually in response to an error, or by the programmer. By default, IntraBuilder handles an exception by displaying an error dialog and terminating the currently executing script. You can use *finally* to make sure some code gets executed even if there is an exception, and *catch* to handle the exception yourself, in the following combinations:

- For a block of code that may generate an exception, place the code inside a *try* block. To prevent the exception from generating a standard error dialog and terminating execution, place exception handling code in a *catch* block after the *try*. If an exception occurs, execution immediately jumps to the *catch* block; no more statements in the *try* block are executed. If no exception occurs, the *catch* block is not executed.
- If there's some code that should always be executed at the end of a process, whether or not the process completes successfully, place that code in a *finally* block. With *try* and *finally* but no *catch*, if an exception occurs during the *try* block, execution immediately jumps to the *finally* block; no more statements in the *try* block are executed. Since there was no *catch*, you would still have an exception, which if not handled by a higher-level *catch* as described later, IntraBuilder would handle as usual, after executing the *finally* block. If no exception occurs, the *finally* block is executed after the *try*.
- If you have all three—*try*, *catch*, and *finally*—if an exception occurs, execution immediately jumps to the *catch* block; after the *catch* block executes, the *finally* block is executed. If there is no exception during the *try*, then the *catch* block is skipped, and the *finally* block is executed.

The code that is covered by *try* doesn't have to be inside the statement block physically; the coverage exists until that entire block of code is executed. For example, you may have a function call inside a *try*

block, and if an exception occurs while that function is executing—even if that function is defined in another script file—execution jumps back to the corresponding *catch* or *finally*.

A *try* block may be followed by multiple *catch* blocks, each with its own <exception type>. When an exception occurs, IntraBuilder compares the <exception type> with the *className* property of the Exception object. If they match, that *catch* block is executed and all others are skipped. If the *className* does not match, IntraBuilder searches the class hierarchy of that object to find a match. If no match is found, the next *catch* block is tested. For example, the DbException class is a subclass of the Exception class. If the blocks are arranged like this:

```
try {
  // Statements
}
catch ( DbException e ) {
  // Block 1
}
catch ( Exception e ) {
  // Block 2
}
```

then if a DbException occurs, execution goes to Block 1, because that's a match. If an Exception occurs, execution goes to Block 2, because Block 1 doesn't match, but Block 2 does. If the blocks are arranged the other way around, like this:

```
try {
  // Statements
}
catch ( Exception e ) {
  // Block 1
}
catch ( DbException e ) {
  // Block 2
}
```

then all exceptions always go to Block 1, because all Exceptions are derived from the Exception class. Therefore, when using multiple *catch* blocks, list the most specific exception classes first.

You can generate exceptions on purpose with the *throw* statement to control program flow. For example, if you enter deeply nested control structures or subroutines from a *try* block, you can *throw* an exception from anywhere in the nested code. This would cause execution to jump back to the corresponding *catch* or *finally*, instead of having to exit each control structure or subroutine one-by-one.

You may also nest *try* structures. An exception inside the *try* block causes execution to jump to the corresponding *catch* or *finally*, but an exception in a *catch* or *finally* is simply treated as an exception. Also, if you have a *try* and *finally* but no *catch*, that leaves you with an unhandled exception. If the *try*/*catch*/*finally* is itself inside a *try* block, then that exception would be handled at that next higher level, as illustrated in the following code skeleton:

```
try {
  // exception level 1
  try {
    // exception level 2
  }
  catch ( Exception e ) {
    // handler for level 2
    // but exception level 1
  }
  finally {
    // level 2
  }
}
catch ( Exception e ) {
```

```
  // handler for level 1
}
```

Note that if an exception occurs in the level 2 *catch*, the level 2 *finally* is still executed before going to the level 1 *catch*, because a *finally* block is always executed after a *try* block.

In addition to exceptions, other program flow deviations—specifically *break*, *continue*, and *return*—are also caught by *try*. If there is a corresponding *finally* block, it's executed before control is transferred to the expected destination. (*catch* catches only exceptions.)

try example

The following example illustrates how to code a transaction, which is an all-or-nothing attempt at multiple database changes. If any of the changes should fail—for example, attempting to write a new record to disk, which would fail if there was no more disk space—the entire transaction must be rolled back. In the example, *this* refers to a button on a form:

```
try {
  this.form.rowset.parent.database.beginTrans(); // begin the transaction
  //
  // make changes
  //
  this.form.rowset.parent.database.commit();     // if you got this far,
there were no
                                                 // errors, so commit the
changes
}
catch ( Exception e ) {  // the parameter receives the Exception object that
describes
                         // the error (not used in this example, but
required)
  this.form.rowset.parent.database.rollback();   // undo any changes that did
take
  // display an error message
}
```

This example runs a process in a subdirectory, the name of which is passed as the parameter cDir. It uses two *try* blocks to create the subdirectory if necessary, and return to the previous directory even if there is an error in the process:

```
try {
  // Instead of bothering to see if the directory already exists
  _sys.os.makeDir( cDir );    // Go ahead and try to create the directory
}
catch ( Exception e ) {  // If there's an error creating the directory,
                         //   execution goes here.
  // Do nothing -- this assumes the error is because the directory already
exists.
  // By using a catch, the error is ignored.
}
finally {
  _sys.os.changeDir( cDir );  // Now try and go to that directory
  // At this point, if you can't go to the directory, then that's a real
error.
  // That would be handled normally, since the error occurred in the finally
and
  // is not nested inside another try.
}
try {
  //
  // Run the process
  //
}
// No catch, so if there's an error, there will be a dialog
finally {
  // But because of this finally, the previous directory will be restored
regardless.
```

```
  // This makes the code easier to re-test, since you don’t have to switch
back to
  // your main directory manually after an error.
  _sys.os.changeDir( ".." );
}
```

unescape()

Related topics Example

Decodes a string encoded with *escape*().

Syntax

unescape(<expC>)

<expC>

The character expression to decode.

Description

Use *unescape*() to decode a character string that has been encoded with the *escape*() function. Character strings are encoded to replace unsafe characters. Most non-alphanumeric characters, like spaces and most punctuation marks, are considered unsafe; they cannot be used in URLs. By using *escape*() to encode those characters, the value of the string can be passed in a URL in its encoded form. Then the *unescape*() function can be used to decode the string to get its actual value.

The encoding for a character is its ASCII value in the ISO Latin-1 character set in hexadecimal, preceded by the "%" character. For example,

```
escape( "hi!" )    // returns "hi%21"
escape( "a,b" )    // returns "a%2Cb"
```

Whenever *unescape*() encounters a "%" character in <expC>, it attempts to interpret the next two characters as a hexadecimal ASCII value to determine the actual value of the encoded character. It returns all other characters unchanged.

unescape() example

The following custom Header for the report SEARCH.JRP takes the page to display as the first parameter and an SQL statement as the second parameter. The link to display the next page in the report, which is managed by the report's *isLastPage*() method, must contain the SQL statement. Because the SQL statement contains spaces, it cannot be used as-is in the URL; therefore it is encoded with the *escape*() function before it is stored.

When the report is called with the encoded SQL statement, the *unescape*() function is used to decode the string. To determine whether the SQL statement is encoded, the *indexOf*() method is used; if the string contains a space, it is not encoded.

```
var r = new SEARCHReport();
if (SEARCH.arguments.length == 2) {
  var sql = SEARCH.arguments[1];
  if ( sql.indexOf( " " ) >= 2) {
    // If there's a space, it's a plain SQL statement
    r.query1.sql = sql;
    r.encodedSQL = escape( sql );
  }
  else {
    // otherwise, it's encoded
    r.query1.sql = unescape( sql );
    r.encodedSQL = sql;
  }
}
else {
  r.encodedSQL = "";
}
if (SEARCH.arguments.length >= 1) {
   r.startPage = r.endPage = SEARCH.arguments[0];
}
r.render();
return;
```

var

Example

Declares and optionally initializes variables that you can use in the routine where they're declared and in all lower-level subroutines.

Syntax

var <variable> [= <value>] [, <variable> [= <value>] [, ...]]

<variable>

The name of the variable. Although IntraBuilder imposes no limit to the length of variable names, it recognizes only the first 32 characters.

<value>

An optional value to assign to the variable.

Description

Use *var* in a function to avoid accidentally overwriting a variable with the same name that was declared in a higher-level routine. Normally, variables are visible and changeable in lower-level routines. In effect, *var* hides any existing variable with the same name that was not created in the current routine. If a value is specified in the *var* statement, that value is assigned to the named variable; if not, the variable is not created by the *var* statement.

It's a good practice to always use *var*, unless you want to create a global variable. For example, if you write a function that someone else might use, you probably won't know what variables they're using. If you don't use *var*, you might accidentally change the value of one of their variables when they call your function.

var example

The function isBlank() creates and uses a work variable named c that is initialized with the value of the parameter cArg:

```
function isBlank( cArg ) {
  var c = cArg;                       // Make a work copy
  while ( c.length > 0 && c.charAt( 0 ) == " " ) {
    c = c.substring( 1, c.length );  // Remove first character (a space)
  }
  return c == "";
}
```

It's possible that the person calling the function already has a variable named c that's doing something else. If you don't declare c with *var*, their copy of c will be overwritten by the one created in this function.

while

Related topics Example

A control statement that executes a statement or block of statements while a specified condition is *true*.

Syntax

while (<condition expL>)
    <statement> | { <statement block> }

<condition expL>

A logical expression that is evaluated before each iteration of the loop to determine whether the iteration should occur. If it evaluates to *true*, the statements are executed. Once it evaluates to *false*, the loop is terminated and execution continues with the statement following the *while* loop.

<statement> | { <statement block> }

A single statement or a statement block inside curly braces that is executed in each iteration of the loop.

Description

Use a *while* loop to repeat a statement or block of statements while a condition is *true*. If the condition is initially *false*, the statements are never executed.

You may also exit the loop with *break*, or restart the loop with *continue*.

while example

The following function determines whether a string is blank by removing all spaces from a string in a loop and seeing if there's anything left. For the loop to execute, the *length* of the string must be greater than zero and the first character must be a space. If either of these conditions is not true, even at the beginning of the loop, no characters are removed.

```
function isBlank( cArg ) {
  var c = cArg;                       // Make a work copy
  while ( c.length > 0 && c.charAt( 0 ) == " " ) {
    c = c.substring( 1, c.length );  // Remove first character (a space)
  }
  return c == "";
}
```

with

Example

A control statement that causes all the variable and property references within it to first assume that they are properties of the specified object.

Syntax

with (<oRef>)
    <statement> | { <statement block> }

<oRef>

A reference to the default object.

<statement> | { <statement block> }

A single statement or a statement block inside curly braces that assumes that the specified object is the default.

Description

Use *with* when working with multiple properties of the same object. Instead of using the object reference and the dot operator every time you refer to a property of that object, you specify the object reference once. Then every time a variable or property name is used, it is first checked to see if that name is a property of the specified object. If it is, then that property is used. If not, then the variable or property name is used as is.

with example

The following code from a JFM file assigns values to the properties of a newly created Query object inside a *with* block. Note that the object is created, assigned as a property of the form (referred to by *this* in the excerpted code), and used as the <oRef> for the *with* block, all in one place in the code:

```
with (this.messages1 = new Query()) {
   left = 55.25;
   top = 4.9;
   sql = "select * from MESSAGES.DB";
   active = true;
}
```

Without the *with* block, the code would have looked like this:

```
this.messages1 = new Query();
this.messages1.left = 55.25;
this.messages1.top = 4.9;
this.messages1.sql = "select * from MESSAGES.DB";
this.messages1.active = true;
```

String object

All strings are String objects. Unlike most other objects, String objects have an inherent value: the characters in the strings themselves. The *new* operator is not required to create strings; simply assign a new character expression. A string literal is enclosed in either single or double quotes; for example either ‘Borland’ or “Borland”. Whether a string is a variable, property of an object, literal, or character expression, the methods and properties listed may be applied.

The maximum length of a string in IntraBuilder is approximately 2 billion characters , or the amount of virtual memory on the server, whichever is less. Maximum string lengths in JavaScript-capable browsers is client-dependent.

IntraBuilder has two related string classes: String, which matches the string objects in client-side JavaScript, and StringEx, which includes more string-processing methods.

class String

Example

A string of characters.

Syntax

[<oRef> =] <expC>

or

[<oRef> =] *new* String([<expC>])

<oRef>

A variable or property in which you want to store a reference to the newly created String object.

<expC>

The string you want to create. If omitted, the string is empty.

Properties

The following tables list the properties and methods of the String class. (No events are associated with this class.)

Property	Default	Description
className	String	Identifies the object as an instance of the String class
length		The number of characters in the string
string		The value of the String object

Method	Parameters	Description
anchor()	<expC>	Tags the string as an anchor <A NAME>
big()		Tags the string as big <BIG>
blink()		Tags the string as blinking <BLINK>
bold()		Tags the string as bold <BOLD>
charAt()	<index expN>	Returns the character in the string at the designated position
fixed()		Tags the string as fixed font <TT>
fontcolor()	<expC>	Tags the string as the designated color
fontsize()	<expN>	Tags the string as the designated font size
indexOf()	<expC> [, <start index expN>]	Returns the position of the search string inside the string
italics()		Tags the string as in italics <I>
lastIndexOf()	<expC> [, <start index expN>]	Returns the position of the search string inside the string, searching backwards
link()	<expC>	Tags the string as a link <A HREF>
small()		Tags the string as small <SMALL>
strike()		Tags the string as strikethrough <STRIKE>
sub()		Tags the string as subscript <SUB>
substring()	<start index expN> , <end index expN>	Returns a substring derived from the string
sup()		Tags the string as superscript <SUP>
toLowerCase()		Returns the string in all lowercase
toUpperCase()		Returns the string in all uppercase

Description

A String object contains the actual string value, stored in the property *string*, and methods that act upon that value. The methods do not modify the value of *string*; they use it as a base and return another string

or number.

The methods are divided into two categories: those that simply wrap the string in HTML tags and those that act upon the contents of the string.

Because the return values for most string methods are also strings, you can call more than one method for a particular string by chaining the method calls together. For example,

```
cSomething.substring( 4, 7 ).toUpperCase()
```

class String example

Execute the following statements in the Script Pad (do not type in the comments):

```
cSomething = new String( "Something" )
c = cSomething.substring( 4, 7 )
? c                    // Displays "thi"
? c.toUpperCase()      // Displays "THI"
? "Something".substring( 4, 7 ).toUpperCase()  // Displays "THI"
```

class StringEx

Example

A string of characters with extended capabilities.

Syntax

[<oRef> =] *new* StringEx([<expC>])

<oRef>

A variable or property in which you want to store a reference to the newly created String object.

<expC>

The string you want to create. If omitted, the string is empty.

Properties

The following tables list the properties and methods of the StringEx class. StringEx objects also inherit the properties and methods of the String class. (No events are associated with the either class.)

Property	Default	Description
className	StringEx	Identifies the object as an instance of the StringEx class (Property discussed in Chapter 5, “Core language.”)

Method	Parameters	Description
asc()	<expC>	Returns the ASCII value of the first character in the designated string
chr()	<expN>	Returns the character equivalent of the specified ASCII value
isAlpha()		Returns *true* if the first character of the string is alphabetic
isLower()		Returns *true* if the first character of the string is lowercase
isUpper()		Returns *true* if the first character of the string is uppercase
left()	<expN>	Returns the specified number of characters from the beginning of the string
leftTrim()		Returns the string with all leading spaces removed
replicate()	<expC> [, <expN>]	Returns the specified string repeated a number of times
right()	<expN>	Returns the specified number of characters from the end of the string
rightTrim()		Returns the string with all trailing spaces removed
space()	<expN>	Returns a string comprising the specified number of spaces
stuff()	<start expN> , <quantity expN> [, <replacement expC>]	Returns the string with specified characters removed and others inserted in their place
toProperCase()		Returns the string in proper case

Description

A StringEx object is an extended version of the String object that contains the actual string value, stored in the property *string*, and more methods that act upon that value. The methods do not modify the value of *string*; they use it as a base and return another string or number.

The strings that the StringEx methods return are String objects, not StringEx objects. The string in a StringEx object’s *string* property is also a String object.

The methods are divided into two categories: those that act upon the contents of the string, and utility string functions. The utility functions must be called from a StringEx object.

If you intend to call multiple utility functions, you can create and reuse a StringEx object. For example,

```
var sX = new StringEx();
_sys.scriptOut.writeln( sX.replicate( "*", 10 ) );
_sys.scriptOut.write( sX.space( 10 ) + "!");
```

Or you can create a StringEx object for a *with* block. For example,

```
with ( new StringEx() ) {
  _sys.scriptOut.writeln( replicate( "*", 10 ) );
  _sys.scriptOut.write( space( 10 ) + "!" );
}
```

For a single function call, you can create and use the StringEx object in the same statement:

```
_sys.scriptOut.writeln( new StringEx().asc( "A" ) );
```

class StringEx example

The following function removes characters from a string with the *stuff*() method of the StringEx class. Because the *stuff*() method returns a String object instead of a StringEx object, the StringEx object updates itself by assigning the return value directly into its own *string* property.

```
_sys.scriptOut.writeln( strip( "banana", "an" ) );  // Displays "ba"
function strip( cArg, cStrip )
{
  var cRet = new StringEx( cArg );                  // Make copy as StringEx
object
  var n;                                            // Work variable
  while ( ( n = cRet.indexOf( cStrip ) ) >= 0 ) {   // While string is found
    cRet.string = cRet.stuff( n, cStrip.length );   // Remove it
  }
  return cRet.string;                               // Return a String object
}
```

asc()

Related topics Example

Returns the numeric ASCII value of a specified character.

Syntax

<oRef>.*asc*(<expC>)

<oRef>

A reference to a StringEx object.

<expC>

The character whose ASCII value you want to return. You can specify a string with more than one character, but IntraBuilder uses only the first one.

Property of

StringEx

Description

asc() is the inverse of *chr*(). *asc*() accepts a character and returns its ASCII value—a number from 0 to 255, inclusive. *chr*() accepts an ASCII value and returns its character.

asc() example

Executing the following statements in the Script Pad (do not type the comments) demonstrates the relation between *asc*() and *chr*():

```
sX = new StringEx()
? sX.asc( "A" )                              // 65
? sX.chr( sX.asc( "A" ) + 32 )               // "a"
? sX.asc( sX.chr( sX.asc( "A" ) + 32 ) )     // 97
```

charAt()

Related topics Example

Returns the character at the specified position in the string.

Syntax

<expC>.*charAt*(<expN>)

<expC>

A String or StringEx object.

<expN>

Index into the string, which is indexed from left to right. The first character of the string is at index 0 and the last character is at index <expC>.length – 1.

Property of

String, StringEx

Description

charAt() returns the character in a String or StringEx object at the specified index position. If the index position is after the last character in the string, *charAt*() returns an empty string.

charAt() example

The following client-side JavaScript function determines whether a string is blank by removing all spaces at the end of a string in a loop and seeing if there's anything left. For the loop to execute, the *length* of the string must be greater than zero and the last character, which is extracted with the *charAt*() method, must be a space. If either of these conditions is not true, even at the beginning of the loop, no characters are removed.

```
function isBlank( cArg ) {
  var c = cArg;                          // Make a work copy
  while ( c.length > 0 && c.charAt( c.length - 1 ) == " " ) {
    c = c.substring( 0, c.length - 1 );  // Remove last character (a space)
  }
  return c == "";
}
```

IntraBuilder's StringEx class has a *rightTrim*() method that will remove all trailing spaces with a single call.

chr()

Related topics Example

Returns the character equivalent of a specified ASCII value.

Syntax

<oRef>.*chr*(<expN>)

<oRef>

A reference to a StringEx object.

<expN>

The numeric ASCII value, from 0 to 255, inclusive, whose character equivalent you want to return.

Property of

StringEx

Description

chr() is the inverse of *asc*(). *chr*() accepts an ASCII value and returns its character, while *asc*() accepts a character and returns its ASCII value.

chr() example

Executing the following statements in the Script Pad (do not type the comments) demonstrates the relation between *asc*() and *chr*():

```
sX = new StringEx()
? sX.asc( "A" )                              // 65
? sX.chr( sX.asc( "A" ) + 32 )               // "a"
? sX.asc( sX.chr( sX.asc( "A" ) + 32 ) )     // 97
```

indexOf()

Related topics Example

Returns a number that represents the position of a string within another string.

Syntax

<target expC>.*indexOf*(<search expC> [, <from index expN>])

<target expC>

The string in which you want to search for <search expC>.

<search expC>

The string you want to search for in <target expC>.

<from index expN>

Where you want to start searching for the string. By default, IntraBuilder starts searching at the beginning of the string, index 0.

Property of

String, StringEx

Description

indexOf() returns an index representing where a search string begins in a target string. The first character of the string is at index 0 and the last character is at index <target expC>.length – 1. *indexOf*() searches one character at a time from left to right, beginning to end, from the character at <from index expN> to the last character. The search is case-sensitive. Use *toUpperCase*() or *toLowerCase*() to make the search case-insensitive.

indexOf() returns –1 when

- The search string isn't found.
- The search string or target string is empty.
- The search string is longer than the target string.

Use *lastIndexOf*() to find the starting position of <search expC>, searching from right to left, end to beginning.

indexOf() example

The following function removes characters from a string by looking for the search string with *indexOf*() and replacing it with nothing. Because the *stuff*() method returns a String object instead of a StringEx object, the StringEx object updates itself by assigning the return value directly into its own *string* property.

```
_sys.scriptOut.writeln( strip( "banana", "an" ) );  // Displays "ba"
function strip( cArg, cStrip )
{
  var cRet = new StringEx( cArg );                  // Make copy as StringEx
object
  var n;                                            // Work variable
  while ( ( n = cRet.indexOf( cStrip ) ) >= 0 ) {   // While string is found
    cRet.string = cRet.stuff( n, cStrip.length );   // Remove it
  }
  return cRet.string;                               // Return a String object
}
```

isAlpha()

Related topics

Returns *true* if the first character of a string is alphabetic.

Syntax

<oRef>.*isAlpha*()

<oRef>

A reference to the StringEx object you want to test.

Property of

StringEx

Description

isAlpha() tests the first character of the *string* property in a StringEx object and returns *true* if it's an alphabetic character. *isAlpha*() returns *false* if the character isn't alphabetic or if the character expression is empty.

isLower()

Related topics

Returns *true* if the first character of a string is alphabetic and lowercase.

Syntax

<oRef>.*isLower*()

<oRef>

A reference to the StringEx object you want to test.

Property of

StringEx

Description

isLower() tests the first character of the *string* property in a StringEx object and returns *true* if it's a lowercase alphabetic character. *isLower*() returns *false* if the character isn't lowercase or if the character expression is empty.

isUpper()

Related topics

Returns *true* if the first character of a string is alphabetic and uppercase.

Syntax

<oRef>.*isUpper*()

<oRef>

A reference to the StringEx object you want to test.

Property of

StringEx

Description

isUpper() tests the first character of the *string* property in a StringEx object and returns *true* if it's an uppercase alphabetic character. *isUpper*() returns *false* if the character isn't uppercase or if the character expression is empty.

lastIndexOf()

Related topics Example

Returns a number that represents the starting position of a string within another string. *lastIndexOf*() searches backward from the right end of the target string, and returns a value counting from the beginning of the target.

Syntax

<target expC>.*lastIndexOf*(<search expC> [, <from index expN>])

<target expC>

The string in which you want to search for <search expC>.

<search expC>

The string you want to search for in <target expC>.

<from index expN>

Where you want to start searching for the string. By default, IntraBuilder starts searching at the end of the string, index <target expC>.length – 1.

Property of

String, StringEx

Description

Use *lastIndexOf*() to search for the <search expC> in a target string, searching right to left, end to beginning, from the character at <from index expN> to the first character. The search is case-sensitive. Use *toUpperCase*() or *toLowerCase*() to make the search case-insensitive.

Even though the search starts from the end of the target string, *lastIndexOf*() returns an index representing where a search string begins in a target string, counting from the beginning of the target. The first character of the string is at index 0 and the last character is at index <target expC>.length – 1. If <search expC> occurs only once in the target, *lastIndexOf*() and *indexOf*() return the same value. For example, “abcdef”.*lastIndexOf*(“abc”) and “abcdef”.*indexOf*(“abc”) both return 0.

lastIndexOf() returns –1 when:

- The search string isn’t found
- The search string is an empty string
- The search string is longer than the target string

To find the starting position of <search expC>, searching from left to right, beginning to end, use *indexOf*().

lastIndexOf() example

Here is a simple file name extraction function that extracts a file name for a path by looking for the last backslash character with *lastIndexOf*(). Everything that follows in the string is extracted with *substring*(). If there is no backslash, the entire string is returned.

```
_sys.scriptOut.writeln( getFileName( "C:\\WINDOWS\\SOL.EXE" ) );
function getFileName( cArg )
{
  var n = cArg.lastIndexOf( "\\" );
  return n >= 0 ? cArg.substring( ++n, cArg.length ) : cArg;
}
```

Note that the index position returned by *lastIndexOf*() is incremented before it is passed to *substring*. Otherwise, the last backslash would be returned as well.

left()

Related topics

Returns a specified number of characters from the beginning of a character string.

Syntax

<oRef>.*left*(<expN>)

<oRef>

The string from which you want to extract characters.

<expN>

The number of characters to extract from the beginning of the string.

Property of

StringEx

Description

Starting with the first character of a character expression, *left*() returns <expN> characters. If <expN> is greater than the number of characters in the specified string, *left*() returns the string as it is, without adding space characters to achieve the specified length. You can use the string's *length* property to determine the actual length of the returned string.

If <expN> is less than or equal to zero, *left*() returns an empty string. If <expN> is greater than or equal to zero, *left*(<expC>, <expN>) achieves the same results as *substring*(<expC>, 1, <expN>).

leftTrim()

Related topics

Returns a string with no leading space characters.

Syntax

<oRef>.*leftTrim*()

<*oRef***>**

A reference to the StringEx object from which you want to remove the leading space characters.

Property of

StringEx

Description

leftTrim() returns the *string* property in a StringEx object as a character expression with no leading space characters.

To remove trailing space characters from a string, use *rightTrim*().

length [string]

Related topics

The number of characters in a specified character string.

Syntax

<expC>.*length*

<expC>

The character string whose length you want to find.

Property of

String, StringEx

Description

A string's *length* property reflects the number of characters (the length) of a character string. The length of an empty character string is zero. *length* counts an embedded null character as one character.

replicate()

Related topics Example

Returns a string repeated a specified number of times.

Syntax

<oRef>.*replicate*(<expC> [, <expN>])

<*oRef***>**

A reference to a StringEx object.

<expC>

The string you want to repeat.

<expN>

The number of times to repeat the string; by default, 1.

Property of

String, StringEx

Description

replicate() returns a character string composed of a character expression repeated a specified number of times.

If the character expression is an empty string, *replicate*() returns an empty string. If the number of repeats you specify for <expN> is 0, *replicate*() returns an empty string. If <expN> is less than 0, IntraBuilder displays an error.

To repeat space characters, use *space*().

replicate() example

The following statement executed in the Script Pad displays 10 dollar signs:

```
? new StringEx().replicate( "$", 10 )
```

right()

Related topics

Returns characters from the end of a character string.

Syntax

<oRef>.*right*(<expN>)

<*oRef***>**

A reference to the StringEx object from which you want to extract characters.

<expN>

The number of characters to extract from the string.

Property of

StringEx

Description

Starting with the last character of the *string* property in a StringEx object, *right*() returns a specified number of characters. If the number of characters you specify for <expN> is greater than the number of characters in the specified string, *right*() returns the string as is, without adding space characters to achieve the specified length. If <expN> is less than or equal to zero, *right*() returns an empty string.

rightTrim()

Related topics

Returns a string with no trailing space characters.

Syntax

<oRef>.*rightTrim*()

<oRef>

A reference to the StringEx object from which you want to remove the trailing space characters.

Property of

StringEx

Description

rightTrim() returns the *string* property in a StringEx object with no trailing space characters.

To remove leading space characters from a string, use *leftTrim*().

space()

Related topics

Returns a specified number of space characters.

Syntax

oRef.*space*(<expN>)

<oRef>

A reference to a StringEx object.

<expN>

The number of spaces you want to return.

Property of

StringEx

Description

space() returns a character string composed of a specified number of space characters. The space character is ASCII code 32.

If <expN> is 0, *space*() returns an empty string. If <expN> is less than 0, IntraBuilder displays an error.

To create a string using a character other than the space character, use *replicate*().

stuff()

Related topics Example

Returns a string with specified characters removed and others inserted in their place.

Syntax

<oRef>.*stuff*(<start expN>, <quantity expN> [, <replacement expC>])

<oRef>

A reference to the StringEx object in which you want to remove and replace characters.

<start expN>

The character position in the string at which you want to start removing characters.

<quantity expN>

The number of characters you want to remove from the string.

<replacement expC>

The characters you want to insert in the string. By default, this is an empty string.

Property of

StringEx

Description

stuff() returns the string property in a StringEx object with a replacement character string inserted at a specified position. Starting at the position you specify, <start expN>, *stuff*() removes a specified number, <quantity expN>, of characters from the original string.

If the target character expression is an empty string, *stuff*() returns the replacement string.

If <start expN> is less than 0, *stuff*() treats <start expN> as 0. If <quantity expN> is less than or equal to 0, *stuff*() inserts the replacement string at position <start expN> without removing any characters from the target.

If <start expN> is greater than the length of the target, *stuff*() doesn't remove any characters and appends the replacement string to the end of the target.

If the replacement string is empty, *stuff*() removes the characters specified by <quantity expN> from the target, starting at <start expN>, without adding characters.

stuff() example

The following function removes characters from a string with *stuff*() by looking for the search string and replacing it with nothing. Because the *stuff*() method returns a String object instead of a StringEx object, the StringEx object updates itself by assigning the return value directly into its own *string* property.

```
_sys.scriptOut.writeln( strip( "banana", "an" ) );  // Displays "ba"
function strip( cArg, cStrip )
{
  var cRet = new StringEx( cArg );                  // Make copy as StringEx
object
  var n;                                            // Work variable
  while ( ( n = cRet.indexOf( cStrip ) ) >= 0 ) {   // While string is found
    cRet.string = cRet.stuff( n, cStrip.length );   // Remove it
  }
  return cRet.string;                               // Return a String object
}
```

substring()

Related topics Example

Returns a substring derived from a specified character string.

Syntax

<expC>.*substring*(<index1 expN>, <index2 expN>)

<expC>

The string you want to extract characters from.

<index1 expN>, <index2 expnN>

Indexes into the string, which is indexed from left to right. The first character of the string is at index 0 and the last character is at index <expC>.length – 1.

Property of

String, StringEx

Description

<index1 expN> and <index2 expN> determine the position of the substring to extract. *substring*() begins at the lesser of the two indexes and extracts up to the character before the other index. If the two indexes are the same, *substring*() returns an empty string.

If the starting index is after the last character in the string, *substring*() returns an empty string.

substring() example

Here is a simple file name extraction function that extracts a file name for a path by looking for the last backslash character with *lastIndexOf*(). Everything that follows in the string is extracted with *substring*(). If there is no backslash, the entire string is returned.

```
_sys.scriptOut.writeln( getFileName( "C:\\WINDOWS\\SOL.EXE" ) );
function getFileName( cArg )
{
  var n = cArg.lastIndexOf( "\\" );
  return n >= 0 ? cArg.substring( ++n, cArg.length ) : cArg;
}
```

Note that the index position returned by *lastIndexOf*() is incremented before it is passed to *substring*. Otherwise, the last backslash would be returned as well.

toLowerCase()

Related topics

Converts all uppercase characters in a string to lowercase and returns the resulting string.

Syntax

<expC>.*toLowerCase*()

<expC>

The character string you want to convert to lowercase.

Property of

String, StringEx

Description

toLowerCase() converts the uppercase alphabetic characters in a character expression to lowercase. *toLowerCase*() ignores digits and other characters.

toProperCase()

Related topics

Converts a character string to proper-noun format and returns the resulting string.

Syntax

<oRef>.*toProperCase*()

<*oRef***>**

A reference to the StringEx object you want to convert to proper-noun format.

Property of

StringEx

Description

toProperCase() returns the *string* property in a StringEx object with the first character in each word capitalized and the remaining letters set to lowercase. *toProperCase*() changes the first character of a word only if it is a lowercase alphabetic character.

toUpperCase()

Related topics

Converts all lowercase characters in a string to uppercase and returns the resulting string.

Syntax

<expC>.*toUpperCase*()

<expC>

The character string you want to convert to uppercase.

Property of

String, StringEx

Description

toUpperCase() converts the lowercase alphabetic characters in a character expression to uppercase. *toUpperCase*() ignores digits and other characters.

Math

The global object containing methods and properties for mathematical functions and constants.

Syntax

The *Math* object is automatically created when you start IntraBuilder.

Properties

The following tables list the properties and methods of the *Math* object. (No events are associated with the *Math* object.)

Property	Default	Description
className	Math	Identifies the object as an instance of the Math class
E	2.718281828459	The approximate value of e, the base of the system of natural logarithms
LN2	0.69314718	The approximate value of the natural logarithm of 2
LN10	2.302585	The approximate value of the natural logarithm of 10
LOG2E	1.442695	The approximate value of the base-2 logarithm of e
LOG10E	0.4342944819	The approximate value of the base-10 logarithm of e
PI	3.14159265359	The approximate value of pi, the ratio of a circle's circumference to its diameter
SQRT1_2	0.707106781	The approximate value of the square root of one-half
SQRT2	1.414213562373	The approximate value of the square root of two

Method	Parameters	Description
abs()	<expN>	Returns the absolute value of a specified number
acos()	<expN>	Returns the inverse cosine (arccosine) of a number
asin()	<expN>	Returns the inverse sine (arcsine) of a number
atan()	<expN>	Returns the inverse tangent (arctangent) of a number
atn2()	<sine expN>, <cosine expN>	Returns the inverse tangent (arctangent) of a given point
ceil()	<expN>	Returns the nearest integer that is greater than or equal to a specified number
cos()	<expN>	Returns the trigonometric cosine of an angle
dtor()	<expN>	Returns the radian value of an angle whose measurement is given in degrees
exp()	<expN>	Returns e raised to a specified power
floor()	<expN>	Returns the nearest integer that is less than or equal to a specified number
int()	<expN>	Returns the integer portion of a specified number
log()	<expN>	Returns the logarithm to the base e (natural logarithm) of a specified number
max()	<expN1>, <expN2>	Compares two numbers and returns the greater value
min()	<expN1>, <expN2>	Compares two numbers and returns the lesser value
pow()	<base expN>, <exponent expN>	Returns a number raised to the specified power
random()	[<expN>]	Returns a pseudo-random number between 0 and 1 exclusive (never 0 and never 1)
round()	<expN>	Returns a specified number rounded to the nearest integer
rtod()	<expN>	Returns the degree value of an angle measured in radians

sin()	<expN>	Returns the trigonometric sine of an angle
sqrt()	<expN>	Returns the square root of a number
tan()	<expN>	Returns the trigonometric tangent of an angle

Description

The built-in *Math* object contains properties and methods to perform mathematical calculations. Unlike strings, which are themselves string objects, numbers themselves are not *Math* objects.

When using more than one *Math* object property or method for a calculation or series of calculations, you can use the with statement so that you don't have to repeatedly type Math. For example,

```
with ( Math ) {
  x = rtod( acos(-1) );        //  180.00
  y = rtod( acos( 0) );        //   90.00
  z = rtod( acos( 1) );        //    0.00
}
```

abs()

Related topics

Returns the absolute value of a specified number.

Syntax

Math.*abs*(<expN>)

<expN>

The number whose absolute value you want to return.

Description

abs() returns the absolute value of a number. The absolute value of a number represents its magnitude. Magnitude is always expressed as a positive value, so the absolute value of a negative number is its positive equivalent.

acos()

Related topics

Returns the inverse cosine (arccosine) of a number.

Syntax

Math.*acos*(<expN>)

<expN>

The cosine of an angle, from –1 to +1.

Description

acos() returns the radian value of the angle whose cosine is <expN>. *acos*() returns a number from 0 to pi radians. *acos*() returns zero when <expN> is 1. For values of x from 0 to pi, *acos*(y) returns x if *cos*(x) returns y.

To convert the returned radian value to degrees, use *rtod*(). For example, *acos*(.5) returns 1.05 radians while *rtod*(*acos*(.5)) returns 60.00 degrees.

To find the arcsecant of a value, use the arccosine of 1 divided by the value. For example, the arcsecant of 2 is *acos*(1/2), or 1.05 radians.

asin()

Related topics

Returns the inverse sine (arcsine) of a number.

Syntax

Math.*asin*(<expN>)

<expN>

The sine of an angle, from –1 to +1.

Description

asin() returns the radian value of the angle whose sine is <expN>. *asin*() returns a number from –pi/2 to pi/2 radians. *asin*() returns zero when <expN> is 0. For values of x from –pi/2 to pi/2, *asin*(y) returns x if *sin*(x) returns y.

To convert the returned radian value to degrees, use *rtod*(). For example, *asin*(.5) returns .52 radians while *rtod*(*asin*(.5)) returns 30.00 degrees.

To find the arccosecant of a value, use the arcsine of 1 divided by the value. For example, the arccosecant of 1.54 is *asin*(1/1.54), or .71 radians.

atan()

Related topics

Returns the inverse tangent (arctangent) of a number.

Syntax

Math.*atan*(<expN>)

<expN>

Any positive or negative number representing the tangent of an angle.

Description

atan() returns the radian value of the angle whose tangent is <expN>. *atan*() returns a number from –pi/2 to pi/2 radians. *atan*() returns 0 when <expN> is 0. For values of x from –pi/2 to pi/2, *atan*(y) returns x if *tan*(x) returns y.

To convert the returned radian value to degrees, use *rtod*(). For example, *atan*(1) returns 0.79 radians, while *rtod*(*atan*(1)) returns 45.00 degrees.

atan() differs from *atn2*() in that *atan*() takes the tangent as the argument, but *atn2*() takes the sine and cosine as the arguments.

To find the arccotangent of a value, subtract the arctangent of the value from pi/2. For example, the arccotangent of 1.73 is *PI*/2 – *atan*(1.73), or .52.

atn2()

Related topics

Returns the inverse tangent (arctangent) of a given point.

Syntax

Math.*atn2*(<sine expN>, <cosine expN>)

<sine expN>

The sine of an angle. If <sine expN> is 0, <cosine expN> can't also be 0.

<cosine expN>

The cosine of an angle. If <cosine expN> is 0, <sine expN> can't also be 0. When <cosine expN> is 0 and <sine expN> is a positive or negative (nonzero) number, *atn2*() returns +pi/2 or –pi/2, respectively.

Description

atn2() returns the angle size in radians when you specify the sine and cosine of the angle. *atn2*() returns a number from –pi to +pi radians. *atn2*() returns 0 when <sine expN> is 0. When you specify 0 for both arguments, IntraBuilder returns an error.

To convert the returned radian value to degrees, use *rtod*(). For example, *atn2*(1,0) returns 1.57 radians while *rtod*(*atn2*(1,0)) returns 90.00 degrees.

atn2() differs from *atan*() in that *atn2*() takes the sine and cosine as the arguments, but *atan*() takes the tangent as the argument. See *atan*() for instructions on finding the arccotangent.

ceil()

Related topics

Returns the nearest integer that is greater than or equal to a specified number.

Syntax

Math.ceil(<expN>)

<expN>

A number, from which to determine and return the integer that is greater than or equal to it.

Description

ceil() returns the nearest integer that is greater than or equal to <expN>; in effect, rounding positive numbers up and negative numbers down towards zero. If you pass a number with any digits other than 0 as decimal digits, *ceil*() returns the nearest integer that is greater than the number. If you pass an integer to *ceil*(), or a number with only 0s for decimal digits, it returns the integer equal to the number.

For example,

- *ceil*(2.10) returns 3.00
- *ceil*(–2.10) returns –2.00
- *ceil*(2.00) returns 2.00
- *ceil*(2) returns 2
- *ceil*(–2.00) returns –2.00

See the table in the description of *int*() that compares *int*(), *floor*(), *ceil*(), and *round*().

cos()

Related topics

Returns the trigonometric cosine of an angle.

Syntax

Math.*cos*(<expN>)

<expN>

The size of the angle in radians. To convert an angle's degree value to radians, use *dtor*(). For example, to find the cosine of a 30-degree angle, use *cos*(*dtor*(30)).

Description

cos() calculates the ratio between the side adjacent to an angle and the hypotenuse in a right triangle. *cos*() returns a number from –1 to +1. *cos*() returns 0 when <expN> is pi/2 or 3*pi/2 radians.

The secant of an angle is the reciprocal of the cosine of the angle. To return the secant of an angle, use 1/*cos*().

dtor()

Related topics

Returns the radian value of an angle whose measurement is given in degrees.

Syntax

Math.*dtor*(<expN>)

<expN>

A negative or positive number that is the size of the angle in degrees.

Description

dtor() converts the measurement of an angle from degrees to radians. To convert degrees to radians, IntraBuilder

- Multiplies the number of degrees by pi
- Divides the result by 180
- Returns the quotient

A 180-degree angle is equivalent to pi radians.

Use *dtor*() in the trigonometric functions *sin*(), *cos*(), and *tan*() because these functions require the angle value in radians. For example, to find the sine of a 45-degree angle, use *sin*(*dtor*(45)), which returns .71.

E

Related topics

The approximate value of e, the base of the system of natural logarithms

Syntax

Math.E

Description

E contains a number that is approximately 2.718281828459. e is a constant that can be used in mathematical calculations.

exp()

Related topics

Returns e raised to a specified power.

Syntax

Math.*exp*(<expN>)

<expN>

The positive, negative, or zero power (exponent) to raise the number e to.

Description

exp() returns a number equal to e (the base of the natural logarithm) raised to the <expN> power. For example, *exp*(2) returns 7.39 because *pow*(*E*,2) = 7.39.

exp() is the inverse of *log*(). In other words, if y = *exp*(x), then *log*(y) = x.

floor()

Related topics

Returns the nearest integer that is less than or equal to a specified number.

Syntax

Math.*floor*(<expN>)

<expN>

A number, from which to determine and return the integer that is less than or equal to it.

Description

floor() returns the nearest integer that is less than or equal to <expN>; in effect, rounding positive numbers down and negative numbers up away from zero. If you pass a number with any digits other than zero (0) as decimal digits, *floor*() returns the nearest integer that is less than the number. If you pass an integer to *floor*(), or a number with only zeros for decimal digits, it returns the integer equal to the number.

For example,

- *floor*(2.10) returns 2.00
- *floor*(–2.10) returns –3.00
- *floor*(2.00) returns 2.00
- *floor*(2) returns 2
- *floor*(–2.00) returns –2.00

When you pass a positive number to it, *floor*() operates exactly like *int*(). See the table in the description of *int*() that compares *int*(), *floor*(), *ceil*(), and *round*().

int()

Related topics

Returns the integer portion of a specified number.

Syntax

Math.*int*(<expN>)

<expN>

A number whose integer value you want to determine and return.

Description

Use *int*() to remove the decimal digits of a number and retain only the integer portion, the whole number.

If you pass a number with decimal places to a function or method that uses an integer as an argument, IntraBuilder automatically truncates that number, in which case you don't need to use *int*().

The following table compares *int*(), *floor*(), *ceil*(), and *round*().

<expN>	***int*()**	***floor*()**	***ceil*()**	***round*()**
2.56	2	2	3	3
–2.56	–2	–3	–2	–3
2.45	2	2	3	2
–2.45	–2	–3	–2	–2

LN2

Related topics

The approximate value of the natural logarithm of 2.

Syntax
Math.LN2

Description

LN2 contains a number that is approximately 0.69314718. It can be derived using *log*(2), but it is available as a constant for convenience in mathematical calculations.

LN10

Related topics

The approximate value of the natural logarithm of 10.

Syntax
Math.LN10

Description

LN10 contains a number that is approximately 2.302585. It can be derived using *log*(10), but it is available as a constant for convenience in mathematical calculations.

log()

Related topics

Returns the logarithm to the base e (natural logarithm) of a specified number.

Syntax

Math.*log*(<expN>)

<expN>

A positive nonzero number that equals e raised to the log. If you specify 0 or a negative number for <expN>, IntraBuilder generates an error.

Description

log() returns the natural logarithm of <expN>. The natural logarithm is the power (exponent) to which you raise the mathematical constant e to get <expN>. For example, *log*(5) returns 1.61 because *pow*(*E*,1.61) = 5.

log() is the inverse of *exp*(). In other words, if *log*(y) = x, then y = *exp*(x).

LOG2E

Related topics

The approximate value of the base-2 logarithm of e.

Syntax

Math.LOG2E

Description

LOG2E contains a number that is approximately 1.442695. Use it in conjunction with *log*() to calculate the base-2 logarithms for a given number. Multiply the value returned by *log*() with *LOG2E* to get the base-2 logarithm. For example,

```
with ( Math ) {
  x = log(     2 ) * LOG2E;  //  1.00
  y = log(    32 ) * LOG2E;  //  5.00
  z = log( 65536 ) * LOG2E;  // 16.00
}
```

LOG10E

Related topics

The approximate value of the base-10 logarithm of e.

Syntax

Math.LOG10E

Description

LOG10E contains a number that is approximately 0.4342944819. Use it in conjunction with *log*() to calculate the base-10 logarithms for a given number. Multiply the value returned by *log*() with *LOG10E* to get the base-10 logarithm. For example,

```
with ( Math ) {
  x = log(     1 ) * LOG10E;  // 0.00
  y = log(   100 ) * LOG10E;  // 2.00
  z = log( 10000 ) * LOG10E;  // 4.00
}
```

max()

Related topics

Compares two numbers and returns the greater value.

Syntax

Math.*max*(<expN1>, <expN2>)

<expN1>

A number to compare to a second number.

<expN2>

A number to compare to <expN1>.

Description

Use *max*() to compare two numbers to determine the greater of the two values. You can use *max*() to ensure that a number is not less than a particular limit.

If <expN1> and <expN2> are equal, *max*() returns their value.

min()

Related topics

Compares two numbers and returns the lesser value.

Syntax

Math.*min*(<expN1>, <expN2>)

<expN1>

A number to compare to a second number.

<expN2>

A number to compare to <expN1>.

Description

Use *min*() to compare two numbers to determine the lesser of the two values. You can use *min*() to ensure that a number is not greater than a particular limit.

If <expN1> and <expN2> are equal, *min*() returns their value.

PI

Related topics

The approximate value of pi, the ratio of a circle's circumference to its diameter.

Syntax

Math.PI

Description

PI contains a number that is approximately 3.141592653589793. pi is a constant that can be used in mathematical calculations. For example, use it to calculate the area and circumference of a circle or the volume of a cone or cylinder.

pow()

Related topics

Returns a number raised to the specified power.

Syntax

Math.*pow*(<base expN>,<exponent expN>)

<base expN>

The number you want to raise to the specified <exponent expN>.

<exponent expN>

The exponent or power you want to raise the <base expN>.

Description

pow() returns a base number raised to a specified exponent.

random()

Related topics

Returns a pseudo-random number between 0 and 1 exclusive (never 0 and never 1).

Syntax

Math.*random*([<expN>])

<expN>

The number with which you want to seed *random*(). This option is not available with client-side JavaScript.

Description

Computers cannot generate truly random numbers, but you can use *random*() to generate a series of numbers that appear to have a random distribution. A series of pseudo-random numbers relies on a seed value, which determines the exact numbers that appear in the series. If you use the same seed value, you get the same series of numbers.

Pseudo-random numbers, when considered as a whole series, appear to be random; that is, you cannot tell from one number what the next will be. But the first number in the series is related to the seed value. Therefore, you should seed *random*() only once at the beginning of each series, like before simulating a card shuffle or randomly assigning work shifts. Seeding during a series defeats the design of the random number generator.

If you specify a positive <expN> value, *random*() uses that <expN> as the seed value, so a positive value should be used for testing, since the numbers will be the same each time. If <expN> is negative, *random*() uses a seed value based on the number of seconds past midnight on your computer system clock. As a result, a negative <expN> value most likely will give you a different series of random numbers each time.

If you don't specify <expN>, or use zero, *random*() returns the next number in the series.

When IntraBuilder first starts up, the random number generator is seeded with a fixed internal seed value.

round()

Related topics

Returns a specified number rounded to the nearest integer.

Syntax

Math.*round*(<expN>)

<expN>

The number you want to round.

Description

Use *round*() to round a number to the nearest integer.

If <expN> is halfway between two integers (the fractional portion is exactly .5), the number is rounded toward positive infinity: if the number is positive, it's rounded up; if it is negative, it's rounded down.

For example,

- *round*(2.50) returns 3.00
- *round*(–2.50) returns –2.00
- *round*(2.00) returns 2.00

See the table in the description of *int*() that compares *int*(), *floor*(), *ceil*(), and *round*().

rtod()

Related topics

Returns the degree value of an angle measured in radians.

Syntax

Math.*rtod*(<expN>)

<expN>

A negative or positive number that is the size of the angle in radians.

Description

rtod() converts the measurement of an angle from radians to degrees.

To convert radians to degrees, IntraBuilder

- Multiplies the number of radians by 180
- Divides the result by pi
- Returns the quotient

An angle of pi radians is equivalent to 180 degrees.

Use *rtod*() with the trigonometric functions *acos*(), *asin*(), *atan*(), and *atn2*()
to convert the radian return values of these functions to degrees. For example, *atan*(1) returns the value of the angle in radians, 0.79, while *rtod*(*atan*(1)) returns the value of the angle in degrees, 45.00.

sin()

Related topics

Returns the trigonometric sine of an angle.

Syntax

Math.*sin*(<expN>)

<expN>

The size of the angle in radians. To convert an angle's degree value to radians, use *dtor*(). For example, to find the sine of a 30-degree angle, use *sin*(*dtor*(30)).

Description

sin() calculates the ratio between the side opposite an angle and the hypotenuse in a right triangle. *sin*() returns a number from –1 to +1. *sin*() returns zero when <expN> is zero, pi, or 2pi radians.

The cosecant of an angle is the reciprocal of the sine of the angle. To return the cosecant of an angle, use 1/*sin*().

sqrt()

Related topics

Returns the square root of a number.

Syntax

Math.*sqrt*(<expN>)

<expN>

A positive number whose square root you want to return. If <expN> is a negative number, IntraBuilder generates an error.

Description

sqrt() returns the positive square root of a non-negative number. For example *sqrt*(36) returns 6 because *pow*(6,2) = 36. The square root of 0 is 0.

An alternate way to find the square root is to raise the value to the power of 0.5. For example, the following two statements display the same value:

```
_sys.scriptOut.writeln( Math.sqrt(36) );    // displays 6.00
_sys.scriptOut.writeln( Math.pow(36,.5) );  // displays 6.00
```

SQRT1_2

Related topics

The approximate value of the square root of one-half.

Syntax

Math.SQRT1_2

Description

SQRT1_2 contains a number that is approximately .707106781. It can be derived using *sqrt*(.5), but it is available as a constant for convenience in mathematical calculations.

SQRT2

Related topics

The approximate value of the square root of 2.

Syntax

Math.SQRT2

Description

SQRT2 contains a number that is approximately 1.414213562373. It can be derived using *sqrt*(2), but it is available as a constant for convenience in mathematical calculations.

tan()

Related topics

Returns the trigonometric tangent of an angle.

Syntax

Math.*tan*(<expN>)

<expN>

The size of the angle in radians. To convert an angle's degree value to radians, use *dtor*(). For example, to find the tangent of a 30-degree angle, use *tan*(*dtor*(30)).

Description

tan() calculates the ratio between the side opposite an angle and the side adjacent to the angle in a right triangle. *tan*() returns a number that increases from zero to plus or minus infinity. *tan*() returns zero when <expN> is 0, pi, or 2*pi radians. *tan*() is undefined (returns infinity) when <expN> is pi/2 or 3*pi/2 radians.

The cotangent of an angle is the reciprocal of the tangent of the angle. To return the cotangent of an angle, use 1/*tan*().

Date and time

A Date object represents a moment in time. In JavaScript, it is stored as the number of milliseconds since January 1, 1970 00:00:00 GMT (Greenwich Mean Time). Although GMT and UTC (from the French for Universal Coordinated Time) are derived differently, they are considered to represent the same time in JavaScript.

Because the same moment in time is represented by different times and days on clocks and calendars around the world, and because of the potential international access to your IntraBuilder applications, you must keep time zones in mind.

Modern operating systems have their own current time zone setting, which is used when handling Date objects. For example, two computers with different time zone settings—whether or not they are physically in different time zones—will display the same time differently.

IntraBuilder also features a server-side Timer object that can cause actions to occur at timed intervals.

class Date

Example

An object that represents a moment in time.

Syntax

[<oRef> =] new Date()

or

[<oRef> =] new Date(<date expC>)

or

[<oRef> =] new Date(<msec expN>)

or

[<oRef> =] new Date(<year expN>, <month expN>, <day expN>
[, <hours expN> , <minutes expN> , <seconds expN>])

<oRef>

A variable or property in which you want to store a reference to the newly created Date object.

<date expC>

A string representing a date and time.

<msec expN>

The number of milliseconds since January 1, 1970 00:00:00 GMT. Negative values can be used for dates before 1970.

<year expN>

The year.

<month expN>

A number representing the month, between 0 and 11: zero for January, one for February, and so on, up to 11 for December.

<day expN>

The day of the month, from 1 to 31.

<hours expN>

The hours portion of the time, from 0 to 23.

<minutes expN>

The minutes portion of the time, from 0 to 59.

<seconds expN>

The seconds portion of the time, from 0 to 59.

Properties

The following tables list the properties and methods of the Date class. (No events are associated with this class.)For details on each property, click on the property below.

Property	Default	Description
className	Date	Identifies the object as an instance of the Date class
date		The day of the month
day		The day of the week, from to 0 to 6: 0 is Sunday, 1 is Monday, and so on
hour		The hour of the time
minute		The minute of the time
month		The month of the year, from 0 to 11: 0 is January, 1 is February, and so on

second	The second of the time
year	The year of the date

Method	Parameters	Description
getDate()		Returns day of month
getDay()		Returns day of week
getHours()		Returns hours portion of time
getMinutes()		Returns minutes portion of time
getMonth()		Returns month of year
getSeconds()		Returns seconds portion of time
getTime()		Returns date/time equivalent
getTimezoneOffset()		Returns time zone offset for current locale
getYear()		Returns year of date
parse()	<date expC>	Calculates time equivalent for date string
setDate()	<expN>	Sets day of month
setHours()	<expN>	Sets hours portion of time
setMinutes()	<expN>	Sets minutes portion of time
setMonth()	<expN>	Sets month of year
setSeconds()	<expN>	Sets seconds portion of time
setTime()	<expN>	Sets date/time
setYear()	<expN>	Sets year of date
toGMTString()		Converts date to string, using Internet (GMT) conventions
toLocaleString()		Converts date to string, using locale conventions
toString()		Converts date to string, using standard JavaScript conventions
UTC()	<year expN> , <month expN> , <day expN> [, <hours expN> , <minutes expN> , <seconds expN>]	Calculates time equivalent of date parameters

Event	Parameters	Description

none

Description

A Date object represents both a date and time.

There are four ways to create a new Date object:

- When called with no parameters, the Date object contains the current system date and time.
- You can pass a string containing a date and optionally a time.
- You can pass a number representing the number of milliseconds since January 1, 1970 00:00:00 GMT. Use a negative number for dates before 1970.
- You can pass numeric parameters for each component of the date, and optionally each component of the time.

When you specify a date/time in a string or with the component numbers, the time zone defaults to the current locale.

If you specify a date but don't specify hours, minutes, or seconds, they are set to zero. When passing a string, the <*date expC*> can be in a variety of formats, with or without the time, as shown in the following examples:

```
d1 = new Date( "Jan 5 1996" );           // month, day, year
d2 = new Date( "18 Dec 1994 15:34" );    // day, month, year, and time
d3 = new Date( "1987 Nov 4 9:18:34" );  // year, month, day, and time with
seconds
```

You may spell out the month or abbreviate it, down to the first three letters; for example, “April”, “Apri”, or “Apr”. For consistency and because of the three-letter month of May, you should either always spell it out completely or use the first three letters.

Date objects have an inherent value. The format of the date is platform-dependent; in IntraBuilder, the format is same as using the *toLocaleString*() method. Use the *toGMTString*(), *toLocaleString*(), and *toString*() methods to format the Date objects, or create your own. Date objects will automatically type-convert into strings, using the inherent format.

In IntraBuilder, every Date object has a separate property for each date and time component. You may read or write to these properties directly (except for the *day* property, which is read-only), or use the equivalent method. For example, assigning a value to the *minute* property has the same effect as calling the *setMinutes*() method with the value as the parameter. Client-side JavaScript does not have these properties, so if you want your code to be portable, avoid direct access to the properties and use the methods.

class Date example

The following is an *onServerLoad* event handler for an HTML component on the form. It sets its *text* to the current date and time:

```
function dateLabel_onServerLoad()
{
  this.text = new Date();
}
```

class Timer

Example

An object that initiates a recurring action at preset intervals.

Syntax

[<oRef> =] new Timer()

<oRef>

A variable or property in which you want to store a reference to the newly created Timer object.

Properties

The following tables list the properties and events of the Timer class. (No methods are associated with this class.)For details on each property, click on the property below.

Property	Default	Description
className	Timer	Identifies the object as an instance of the Timer class
enabled	false	Whether the Timer is active
interval	10	The interval between actions, in seconds

Event	Parameters	Description
onTimer		Action to take when *interval* expires

Description

To use a Timer object:

1. Assign an event handler to the *onTimer* event.
2. Set the *interval* property to the desired number of seconds.
3. Set the *enabled* property to *true* when you want to activate the timer.

The Timer object will start counting down time whenever IntraBuilder is idle. When the number of seconds assigned to *interval* has passed, the Timer object's *onTimer* event fires. After the event fires, the Timer object's internal timer is reset back to the *interval*, and the countdown repeats.

To disable the timer, set the *enabled* property to *false*.

There is no way for IntraBuilder to "push" updates to a Web browser; a browser will get updates from an IntraBuilder Agent only if and when they request or submit a form from the browser. Therefore, Timer objects are primarily used for timed automatic processes that you run on the IntraBuilder Designer; not on the IntraBuilder Agent from a browser, although that is possible.

A Timer object counts idle time; that is when IntraBuilder is not doing anything. In the IntraBuilder Designer, this includes waiting for input in the Script Pad or IntraBuilder Explorer. In an IntraBuilder Agent, this is any idle time when the Agent is not servicing a request from a browser. If a process, such as an event handler or script, is running, the counter in all active Timer objects is suspended. When the process is complete and IntraBuilder is idle again, the count resumes.

class Timer example

Suppose you want to display the date and time in a form that you run in the IntraBuilder Designer. The following is an *onServerLoad* event handler that creates a Timer object and attaches it to the form. A reference to the form is added to the Timer object so that the timer's *onTimer* event handler can update the form. Another method in the form is assigned as the Timer object's *onTimer* event handler. The time is updated every two seconds instead of every second, so that IntraBuilder is not too bogged down constantly updating the time.

```
function Form_onServerLoad()
{
  this.timer = new Timer();                 // Make timer a property of the
form
  this.timer.parent   = this;               // Assign form as timer's parent
  this.timer.onTimer  = this.updateClock;   // Assign method in form to timer
  this.timer.interval = 2;                  // Fire timer every 2 seconds
  this.timer.enabled  = true;               // Activate timer
}
```

The following is the updateClock() method of the form, assigned as the *onTimer* event handler. Because the Timer object calls this method, the *this* reference refers to the Timer object, not the form, even though the method is a method of the form. A reference to the form has been stored in the parent property of the timer; an HTML component of the form named clock is updated through that reference.

```
function updateClock()
{
  this.parent.clock.text = new Date();
}
```

The timer should be deactivated when the form is closed. Use the form's *onServerUnload* event:

```
function Form_onServerUnload()
{
  this.timer.enabled = false;
}
```

enabled

Related topics Example

Specifies whether a Timer object is active and counting down time.

Property of

Timer

Description

Set the *enabled* property to *true* to activate the Timer object. When the number of seconds of idle time specified in the *interval* property has passed, the timer's *onTimer* event fires.

When the *enabled* property is set to *false*, the Timer stops counting time and the internal counter is reset. For example, suppose that

1 The *interval* is 10, and *enabled* is set to *true*.

2 Then 9 seconds of idle time go by, and *enabled* is set to *false*.

If *enabled* is set to *true* again, the *onTimer* will fire after another 10 seconds has gone by, even though there was only 1 second left before the timer was disabled.

If a Timer is intended to go off only once instead of repeatedly, set the *enabled* property to *false* in the *onTimer* event handler.

enabled example

Running the following statements in the Script Pad will cause a message to be displayed once, 5 seconds after timer the is enabled:

```
t = new Timer()
t.onTimer = {; ? "Ding!"; this.enabled = false}
t.interval = 5
t.enabled = true
```

getDate()

Related topics Example

Returns the numeric value of the day of the month.

Syntax

<oRef>.*getDate*()

<oRef>

The Date object whose corresponding day-of-the-month number you want to return.

Property of

Date

Description

getDate() returns a date's day of the month number—a value from 1 to 31.

getDate() example

The following is an *onServerLoad* event handler for a form that makes the “New orders” button invisible on the first day of the month, when inventory is being reconciled:

```
function Form_onServerLoad()
{
  if ( new Date().getDate() == 1 ) {      // Get today's day of month
    this.newOrderButton.visible = false;  // Prevent new orders
  }
}
```

getDay()

Related topics Example

Returns the day of the week corresponding to a specified date as a number from 0 to 6.

Syntax

<oRef>.*getDay*()

<oRef>

The Date object whose corresponding weekday number you want to return.

Property of

Date

Description

getDay() returns the number of the day of the week on which a date falls. The number is zero-based:

Day	Number
Sunday	0
Monday	1
Tuesday	2
Wednesday	3
Thursday	4
Friday	5
Saturday	6

The day of the week is the only date/time component you cannot set directly; there is no corresponding set- method. It is always based on the date itself.

getDay() example

The following is an *onServerLoad* event handler for a form that makes the “Game center” button visible on the weekends:

```
function Form_onServerLoad()
{
  if ( new Date().getDay() % 6 == 0 ) {     // If today is a weekend day
    this.gameCenterButton.visible = true;   // Enable access to game center
page
  }
}
```

The day number modulo 6 is zero for both day numbers 0 and 6, the days on the weekend.

getHours()

Related topics Example

Returns the hours portion of a date object.

Syntax

<oRef>.*getHours*()

<oRef>

The date object whose hours you want to return.

Property of

Date

Description

getHours() returns the hours portion of the time (using a 24-hour clock) in a Date object: an integer from 0 to 23.

getHours() example

The following function returns *true* if the date/time passed to it is during the graveyard shift, between 10 p.m. and 6 a.m.:

```
function isGraveyard( dArg )
{
  return ( dArg.getHours() >= 22 || dArg.getHours() < 6 );
}
```

getMinutes()

Related topics Example

Returns the minutes portion of a date object.

Syntax

<oRef>.getMinutes()

<oRef>

The date object whose minutes you want to return.

Property of

Date

Description

getMinutes() returns the minutes portion of the time in a Date object: an integer from 0 to 59.

getMinutes() example

The following is a *preRender* event handler for a form that displays the HTML object minClock as a minute-clock. The *preRender* event fires every time the form is rendered, so the clock is always updated. The clock displays the minutes and seconds only, because the page is designed to be seen in different time zones and all you want to display is the number of minutes (and seconds) past the hour.

```
function Form_preRender()
{
  var dNow = new Date();           // Get time once
  var xMin = dNow.getMinutes();    // Get minutes
  var xSec = dNow.getSeconds();    // and seconds
  if ( xMin < 10 ) {               // Add leading zero if needed
    xMin = "0" + xMin;
  }
  if ( xSec < 10 ) {
    xSec = "0" + xSec;
  }
  this.minClock.text = xMin + ":" + xSec;  // Change text
}
```

getMonth()

Related topics Example

Returns the number of the month.

Syntax

<oRef>.*getMonth*()

<oRef>

The Date object whose corresponding month number you want to return.

Property of

Date

Description

getMonth() returns a date's month number. The number is zero-based:

Month	Number
January	0
February	1
March	2
April	3
May	4
June	5
July	6
August	7
September	8
October	9
November	10
December	11

getMonth() example

The following function formats a date in simple month, day, year format, spelling out the month:

```
function mdy( dArg )
{
  var nYear = dArg.getYear();
  if ( nYear < 100 ) {  // Year in 1900s
    nYear += 1900;
  }
return {"January", "February", "March",
          "April"  , "May"     , "June" ,
          "July"   , "August"  , "September",
          "October", "November", "December"}[ dArg.getMonth() ] +
         " " + dArg.getDate() + ", " + nYear;
}
```

At the beginning of the function, the year is stored in the variable nYear. If it's a two-digit year, that means it's a year in the 1900s, so 1900 is added to make it a four-digit year.

The month number that is returned by *getMonth*() is used as an index into the array of months. The entire result string is built in the *return* statement.

getSeconds()

Related topics Example

Returns the seconds portion of a date object.

Syntax

<oRef>.*getSeconds*()

<oRef>

The date object whose seconds you want to return.

Property of

Date

Description

getSeconds() returns the seconds portion of the time in a Date object: an integer from 0 to 59.

getSeconds() example

The following is a *preRender* event handler for a form that displays the HTML object minClock as a minute-clock. The *preRender* event fires every time the form is rendered, so the clock is always updated. The clock displays the minutes and seconds only, because the page is designed to be seen in different time zones and all you want to display is the number of minutes (and seconds) past the hour.

```
function Form_preRender()
{
  var dNow = new Date();           // Get time once
  var xMin = dNow.getMinutes();    // Get minutes
  var xSec = dNow.getSeconds();    // and seconds
  if ( xMin < 10 ) {               // Add leading zero if needed
    xMin = "0" + xMin;
  }
  if ( xSec < 10 ) {
    xSec = "0" + xSec;
  }
  this.minClock.text = xMin + ":" + xSec;  // Change text
}
```

getTime()

Related topics Example

Returns time equivalent of date/time, in milliseconds.

Syntax

<oRef>.*getTime*()

<oRef>

The Date object whose time equivalent you want to return.

Property of

Date

Description

getTime() returns the number of milliseconds since January 1, 1970 00:00:00 GMT for the date/time stored in the Date object. All date/times are represented internally by this millisecond number.

getTime() example

The following function returns the number of seconds elapsed between two times (in Date objects):

```
function elapsed( dStart, dStop )
{
  return ( dStop.getTime() - dStart.getTime() ) / 1000;
}
```

getTimezoneOffset()

Related topics Example

Returns the time zone offset for a date object in the current locale, in minutes.

Syntax

<oRef>.*getTimezoneOffset*()

<oRef>

A date object created in the locale in question.

Property of

Date

Description

All time zones have an offset from GMT (Greenwich Mean Time), from twelve hours behind to twelve hours ahead. *getTimezoneOffset*() returns this offset, in minutes, for the locale in which the Date object was created, taking Daylight Savings Time into account.

For example, the United States and Canada Pacific time zone is eight hours behind GMT. A date in January, when Daylight Savings Time is not in effect, created in the Pacific time zone would have a time zone offset of –480. A date in July, when Daylight Savings Time is in effect, would have a time zone offset of
–420, or seven hours, since Daylight Savings Time moves clocks one hour forward, closer to GMT.

In Windows, the locale is determined by the Time Zone setting in each system's Date/Time properties, which is found in the Control Panel, or by double-clicking the clock in the Taskbar.

All Date objects default to the time zone setting of the current locale.

getTimezoneOffset() example

The following function determines whether Daylight Savings is active by comparing the current time zone offset with that of a date in January. If they are different, then Daylight Savings is in effect:

```
function isDaylightSavings()
{
  var dTest = new Date();  // Current date
  var nTzO  = dTest.getTimezoneOffset();
  dTest.setMonth( 1 );     // Same date in January
  return ( nTzO != dTest.getTimezoneOffset() );
}
```

This function works only in the northern hemisphere, where January is a winter month—when Daylight Savings is not in effect.

getYear()

Related topics Example

Returns the year of a specified date expression.

Syntax

<oRef>.*getYear*()

<oRef>**<**expD>

The Date object whose corresponding year number you want to return.

Property of

Date

Description

getYear() returns a date's year number.

getYear() example

The following function formats a date in simple month, day, year format, spelling out the month:

```
function mdy( dArg )
{
  var nYear = dArg.getYear();
  if ( nYear < 100 ) {  // Year in 1900s
    nYear += 1900;
  }
return {"January", "February", "March",
          "April"  , "May"     , "June" ,
          "July"   , "August"  , "September",
          "October", "November", "December"}[ dArg.getMonth() ] +
         " " + dArg.getDate() + ", " + nYear;
}
```

At the beginning of the function, the year is stored in the variable nYear. If it's a two-digit year, that means it's a year in the 1900s, so 1900 is added to make it a four-digit year.

The month number that is returned by *getMonth*() is used as an index into the array of months. The entire result string is built in the *return* statement.

interval

Related topics Example

The amount of idle time, in seconds, between the firings of the timer.

Property of

Timer

Description

Set the *enabled* property to *true* to activate the Timer object. When the number of seconds of idle time specified in the *interval* property has passed, the timer's *onTimer* event fires.

When the *enabled* property is set to *false*, the Timer stops counting time and the internal counter is reset. For example, suppose that

1. The *interval* is 10, and *enabled* is set to *true*.
2. Then 9 seconds of idle time go by, and *enabled* is set to *false*.

If *enabled* is set to *true* again, the *onTimer* will fire after another 10 seconds has gone by, even though there was only 1 second left before the timer was disabled.

interval must be zero or greater. The *interval* may be a fraction of a second; the resolution of the timer is one system clock tick, approximately 0.055 seconds. When *interval* is zero, the timer fires once per clock tick.

Setting the *interval* always resets the internal counter to the newly specified time.

interval example

Running the following statements in the Script Pad will cause a message to be displayed once, 5 seconds after timer is enabled:

```
t = new Timer()
t.onTimer = {; ? "Ding!"; this.enabled = false}
t.interval = 5
t.enabled = true
```

onTimer

Related topics Example

When the timer's interval has elapsed.

Parameters

none

Property of

Timer

Description

A Timer object's *onTimer* event is fired every time the amount of idle time specified by the timer's *interval* property has elapsed.

Like all event handlers, inside the *onTimer* event handler, the reference *this* refers to the Timer object itself. To refer to other objects, add references to those objects as properties to the Timer object before activating the timer.

While processing the *onTimer* event, all active timers are suspended, since IntraBuilder is busy processing code. Once the *onTimer* event handler has completed, its internal counter is reset to the *interval*, and all active timers resume counting.

If a Timer is intended to go off only once instead of repeatedly, set the *enabled* property to *false* in the *onTimer* event handler.

onTimer example

Running the following statements in the Script Pad will cause a message to be displayed once, 5 seconds after timer is enabled:

```
t = new Timer()
t.onTimer = {; ? "Ding!"; this.enabled = false}
t.interval = 5
t.enabled = true
```

parse()

Related topics Example

Returns time equivalent of a date/time string, in milliseconds.

Syntax

Date.*parse*(<date expC>)

<date expC>

The date/time string you want to convert.

Property of

Date

Description

parse() returns the number of milliseconds since January 1, 1970 00:00:00 GMT for the specified date/time string, defaulting to the operating system's current time zone setting. For example, if the time zone is currently set to United States Eastern Standard Time, which is five hours behind GMT, then Date.*parse*("Sep 14 1995 11:20") yields a time which is equivalent to 16:20 GMT.

The string may be in any of the forms acceptable to the Date class constructor, as described under class Date. In contrast, the *UTC*() method uses numeric parameters for each of the date and time components and assumes GMT as the time zone.

Because *parse*() is a static class method, you call it via the Date class, not a Date object.

parse() example

The following code fragment resets an existing date object d1 to a date typed into a text control:

```
d1.setTime( Date.parse( this.form.dateText.value ))
```

setDate()

Related topics Example

Sets day of month.

Syntax

<oRef>.*setDate*(<expN>)

<oRef>

The Date object whose day you want to change.

<expN>

The day of month number, normally between 1 and 31.

Property of

Date

Description

setDate() sets the day of month for the Date object. If the day number is greater than the number of days in the Date object's month, then the excess days may cause the date to roll over into the next month. For example,

```
dTest = new Date( "Feb 3 1996" )     // 29 days in month
dTest.setDate( 31 )                  // 2 days over
_sys.scriptOut.writeln( dTest )      // displays Mar 02 1996
dTest = new Date( "Feb 3 1995" )     // 28 days in month
dTest.setDate( 31 )                  // 3 days over
_sys.scriptOut.writeln( dTest )      // displays Mar 03 1995
```

However, this rollover and the maximum day number allowed are implementation-dependent, so test thoroughly on target platforms.

setDate() example

The following function returns the last day of the month of the specified date.

```
#define SECS_PER_HOUR 3600                     // Number of seconds per hour
#define MSECS_PER_DAY (1000*24*SECS_PER_HOUR) // Number of milliseconds per
day
function LDoM( dArg )
{
  var dRet = new Date( dArg.getTime() );           // Make copy of date
argument
  dRet.setDate( 32 );                              // Force date into the
next month
  dRet.setDate( 1 );                               // First day of the next
month
  dRet.setTime( dRet.getTime() - MSECS_PER_DAY );  // Subtract one day
  return dRet;
}
```

Manifest constants created by the #define preprocessor directive are used for the “magic number” of milliseconds per day to make the code easier to read.

setHours()

Related topics Example

Sets hours portion of time.

Syntax

<oRef>.*setHours*(<expN>)

<oRef>

The Date object whose hours you want to change.

<expN>

The hour number, normally between 0 and 23.

Property of

Date

Description

setHours() sets the hours portion of the time for the Date object. If the hour number is greater than 23, then the excess hours may cause the date to roll over into the next day(s). You may need to force the time to be recalculated. For example,

```
dTest = new Date( "Aug 22 1996 21:30" )  // 09:30pm
dTest.setHours( 28 )                     // 4 (=28-24) hours over
dTest.setTime( dTest.getTime() )         // Recalculate time
_sys.scriptOut.writeln( dTest )          // displays Aug 23 1996 04:30am
```

However, this rollover and the maximum hour number allowed are implementation-dependent, so test thoroughly on target platforms.

setHours() example

The following function sets the time of the specified Date object to midnight:

```
function midnight( dArg )
{
  dArg.setHours( 0 );
  dArg.setMinutes( 0 );
  dArg.setSeconds( 0 );
}
```

setMinutes()

Related topics Example

Sets minutes portion of time.

Syntax

<oRef>.*setMinutes*(<expN>)

<oRef>

The Date object whose minutes you want to change.

<expN>

The minute number, normally between 0 and 59.

Property of

Date

Description

setMinutes() sets the minutes portion of the time for the Date object. If the minute number is greater than 59, then the excess minutes may cause the time to roll over into the next hour(s) or day(s). You may need to force the time to be recalculated. For example,

```
dTest = new Date( "Aug 22 1996 23:30" )  // 11:30pm
dTest.setMinutes( 90 )                   // 11:00pm + 1.5 hours = 0.5 hours
over
dTest.setTime( dTest.getTime() )         // Recalculate time
_sys.scriptOut.writeln( dTest )          // displays Aug 23 1996 00:30
```

However, this rollover and the maximum minute number allowed are implementation-dependent, so test thoroughly on target platforms.

setMinutes() example

The following function sets the time of the specified Date object to midnight:

```
function midnight( dArg )
{
  dArg.setHours( 0 );
  dArg.setMinutes( 0 );
  dArg.setSeconds( 0 );
}
```

setMonth()

Related topics Example

Sets month of year.

Syntax

<oRef>.*setMonth*(<expN>)

<oRef>

The Date object whose month you want to change.

<expN>

The month number, normally between 0 and 11: 0 for January, 1 for February, and so on, up to 11 for December.

Property of

Date

Description

setMonth() sets the month of year for the Date object. If the month number is greater than 12, then the excess months may cause the date to roll over into the next year. For example,

```
dTest = new Date( "Apr 9 1995" )
dTest.setMonth( 13 )                // 2 months over
_sys.scriptOut.writeln( dTest )     // displays Feb 09 1996
```

However, this rollover and the maximum month number allowed are implementation-dependent, so test thoroughly on target platforms.

setMonth() example

The following function relies on month rollover to return the date that's a given number of months in the future:

```
function addMonths( dArg, nMonths )
{
  var dRet = new Date( dArg.getTime() );       // Make copy of date argument
  dRet.setMonth( dRet.getMonth() + nMonths );  // Add months to current month
  return dRet;
}
```

setSeconds()

Related topics Example

Sets seconds portion of time.

Syntax

<oRef>.*setSeconds*(<expN>)

<oRef>

The Date object whose seconds you want to change.

<expN>

The number of seconds, normally between 0 and 59.

Property of

Date

Description

setSeconds() sets the seconds portion of the time for the Date object. If the number of seconds is greater than 59, then the excess seconds may cause the time to roll over into the next minute(s), hour(s), or day(s). You may need to force the time to be recalculated. For example,

```
dTest = new Date( "Dec 31 1999 23:59:50" )  // 11:59:50pm
dTest.setSeconds( 60 )                      // 11:59pm + 60 seconds =
12:00mid
dTest.setTime( dTest.getTime() )            // Recalculate time
_sys.scriptOut.writeln( dTest )             // displays Jan 01 2000 00:00:00
```

However, this rollover and the maximum number of seconds allowed are implementation-dependent, so test thoroughly on target platforms.

setSeconds() example

The following function sets the time of the specified Date object to midnight:

```
function midnight( dArg )
{
  dArg.setHours( 0 );
  dArg.setMinutes( 0 );
  dArg.setSeconds( 0 );
}
```

setTime()

Related topics Example

Sets date/time of Date object.

Syntax

<oRef>.setTime(<expN>)

<oRef>

The Date object whose time you want to set.

<expN>

The number of milliseconds since January 1, 1970 00:00:00 GMT for the desired date/time.

Property of

Date

Description

While you may use standard date/time nomenclature when creating a new Date object, *setTime*() requires a number of milliseconds. Therefore *setTime*() is used primarily to copy the date/time from one Date object to another. If you tried copying dates like this:

```
d1 = new Date( "Aug 24 1996" );
d2 = new Date();
d2 = d1;            // Copy date
```

what you're actually doing is copying an object reference for the first Date object into another variable. Both variables now point to the same object, so changing the date/time in one would appear to change the date/time in the other.

To actually copy the date/time, use *setTime*() and *getTime*():

```
d1 = new Date( "Aug 24 1996" );
d2 = new Date();
d2.setTime( d1.getTime() );  // Copy date
```

If you're copying the date/time when you're creating the second Date object, you can use the millisecond value in the Date class constructor:

```
d1 = new Date( "Aug 24 1996" );
d2 = new Date( d1.getTime() );  // Create copy of date
```

You may also perform date math by adding or subtracting milliseconds from the value.

setTime() example

The following function uses date math with *setTime*() to return the last day of the month of the specified date.

```
#define SECS_PER_HOUR 3600                     // Number of seconds per hour
#define MSECS_PER_DAY (1000*24*SECS_PER_HOUR) // Number of milliseconds per
day
function LDoM( dArg )
{
  var dRet = new Date( dArg.getTime() );           // Make copy of date
argument
  dRet.setDate( 32 );                              // Force date into the
next month
  dRet.setDate( 1 );                               // First day of the next
month
  dRet.setTime( dRet.getTime() - MSECS_PER_DAY );  // Subtract one day
  return dRet;
}
```

Manifest constants created by the #define preprocessor directive are used for the “magic number” of milliseconds per day to make the code easier to read.

setYear()

Related topics Example

Sets year of date.

Syntax

<oRef>.*setYear*(<expN>)

<oRef>

The Date object whose year you want to change.

<expN>

The year. For years in the 1900s, you can specify the year as either a 2-digit or 4-digit year.

Property of

Date

Description

setYear() sets the year for the Date object.

setYear() example

The following function adds the specified number of years to a date. It needs to handle the switch between 2-digit and 3-digit years, which if not adjusted would set the date sometime in the dark ages.

```
function addYears( dArg, nYears )
{
  var dRet  = new Date( dArg.getTime() ); // Make copy of date argument
  var nYear = dRet.getYear() + nYears;    // Add years
  if ( nYear > 99 ) {                     // If no longer in 1900s
    nYear += 1900;                        // Add 1900 to get correct year
  }
  dRet.setYear( nYear );                  // Set the new year
  return dRet;
}
```

toGMTString()

Related topics Example

Converts the date into a string, using Internet (GMT) conventions.

Syntax

<oRef>.*toGMTString*()

<oRef>

The Date object you want to convert.

Property of

Date

Description

toGMTString() converts the date, which was created using the operating system's time zone setting, to GMT and returns a string. The exact format of the string depends on the client, for example, "Tue, 07 May 1996 02:55:27 GMT".

toGMTString() example

When the following statement is executed in the Script pad, the current date and time is displayed in the results pane in GMT format:

```
? new Date().toGMTString()
```

toLocaleString()

Related topics Example

Converts the date into a string, using locale conventions.

Syntax

<oRef>.*toLocaleString*()

<oRef>

The Date object you want to convert.

Property of

Date

Description

toLocaleString() converts the date to a string, using the standards for the current locale. The exact format of the string depends on the client, for example, “05/06/96 19:55:27”.

IntraBuilder uses Windows’ Regional settings from the Control Panel.

toLocaleString() example

When the following statement is executed in the Script pad, the current date and time is displayed in the results pane in locale format:

```
? new Date().toLocaleString()
```

toString()

Related topics Example

Converts the date into a string, using standard JavaScript conventions.

Syntax

<oRef>.*toString*()

<oRef>

The Date object you want to convert.

Property of

Date

Description

toString() converts the date to a string, in standard JavaScript format, which includes the complete time zone description, for example,

"Mon May 06 19:55:27 Pacific Daylight Time 1996"

toString() example

When the following statement is executed in the Script pad, the current date and time is displayed in the results pane in standard format:

```
? new Date().toString()
```

UTC()

Related topics Example

Returns time equivalent of the specified date/time parameters using GMT, in milliseconds.

Syntax

Date.*UTC*(<year expN>, <month expN>, <day expN>
[, <hours expN> [, <minutes expN> [, <seconds expN>]]])

<year expN>

The year.

<month expN>

A number representing the month, between 0 and 11: zero for January, one for February, and so on, up to 11 for December.

<day expN>

The day of the month, from 1 to 31.

<hours expN>

The hours portion of the time, from 0 to 23.

<minutes expN>

The minutes portion of the time, from 0 to 59.

<seconds expN>

The seconds portion of the time, from 0 to 59.

Property of

Date

Description

UTC() returns the number of milliseconds since January 1, 1970 00:00:00 GMT for the date/time parameters specified, using GMT as the time zone. In contrast, the *parse*() method takes a string as a parameter, and uses the operating system's current time zone setting as the default.

Because *UTC*() is a static class method, you call it via the Date class, not a Date object.

UTC() example

You cannot specify a time zone when creating a Date object with separate date and time components, but you can use *UTC*() for GMT:

```
dLocale = new Date( nYear, nMonth, nDay );              // Time zone of
locale
dGMT    = new Date( Date.UTC( nYear, nMonth, nDay ) );  // GMT
```

Array objects

IntraBuilder supports a wide variety of array types:

- Arrays of contiguously numbered elements, in one or more dimensions. Elements are numbered from zero. There are methods specifically for one- and two-dimensional arrays, which mimic a row of fields and a table of rows.
- Associative arrays, in which the elements are addressed by a key string instead of a number.
- Sparse arrays, which use non-contiguous numbers to refer to elements.

All arrays are objects, and use square brackets ([ ]) as indexing operators.

Array elements may contain any data type, including object references to other arrays. Therefore you can create nested arrays (multi-dimensional arrays of arrays with fixed length in each dimension), ragged arrays (nested arrays with variable lengths), arrays of associative arrays, and so on.

There are two array classes: Array and AssocArray. Sparse arrays can be created with any other object. In addition to creating properties by name, you can create numeric properties using the indexing operators. For example,

```
o = new Object();
o.title = "Summer";
o[ 2000 ] = "Sydney";
o[ 1996 ] = "Atlanta";
_sys.scriptOut.writeln( o[ 1996 + 4 ] ); // Displays "Sydney"
```

class Array

Related topics Example

An array of elements, in one or more dimensions.

Syntax

[<oRef> =] new Array([<dim1 expN> [,<dim2 expN> ...]])

<oRef>

A variable or property in which to store a reference to the newly created Array object.

<dim1 expN> [,<dim2 expN> ...]

The size of the array in each specified dimension. If no dimensions are specified, the array is a one-dimensional array with zero elements.

Properties

The following tables list the properties and methods of the Array class. (No events are associated with this class.)

Property	Default	Description
className	Array	Identifies the object as an instance of the Array class
dimensions		The number of dimensions in the array
length	0	The number of elements in the array

Method	Parameters	Description
add()	<exp>	Increases the size of a one-dimensional array by one and assigns the passed value to the new element.
delete()	<position expN> [,1 | 2]	Deletes an element from a one-dimensional array, or deletes a row (1) or column (2) of elements from a two-dimensional array, without changing the size of the array.
dir()	[<filespec expC>]	Stores in the array five characteristics of specified files: name, size, modified date, modified time, and DOS attribute(s). Returns the number of files whose characteristics are stored.
dirExt()	[<filespec expC>]	Same as *dir*() method, but adds short (8.3) file name, create date, create time, and access date.
element()	<row expN> [,<col expN>]	Returns the element number for the element at the specified row and column.
fill()	<exp> , <start expN> [, <count expN>]	Stores a specified value into one or more elements of the array.
grow()	1 | 2	When passed 1, adds a single element to a one-dimensional array or a row to a two-dimensional array; when passed 2, adds a column to the array.
insert()	<element expN> [,1 | 2]	Inserts an element, row (1), or column (2) into an array without changing the size of the array (the last element, row, or column is lost).
resize()	<rows expN> [, <cols expN> [, <retain values>]]	Increases or decreases the size of an array. First passed parameter indicates the new number of rows, the second parameter indicates the new number of columns. If the third parameter is zero, current values are relocated; if nonzero, they are retained in their old positions.
scan()	<exp> , <start expN> [, <count expN>]	Searches an array for the specified expression; returns the element number of the first element that matches the expression, or –1 if the search is unsuccessful.
sort()	<start expN> [, <count expN> [, 0 | 1]]	Sorts the elements in a one-dimensional array or the rows in a two-dimensional array in ascending (0) or descending (1) order.
subscript()	<element expN>	Returns the row (1) or column (2) subscript for the specified

 element number.

Description

An Array object is a standard array of elements, addressed by a contiguous range of numbers in one or more dimensions. The array can hold as many elements as memory allows. You can create arrays that contain more than two dimensions, but most IntraBuilder Array methods work only on one- or two-dimensional arrays. For a two-dimensional array, the first dimension is considered the row and the second dimension is the column. For example, the following statement creates an array with 3 rows and 4 columns:

```
a = new Array( 3, 4 );
```

There are two ways to refer to individual elements in an array; you can use either element subscripts or the element number. Element subscripts, one for each dimension, are values that represent the element's position in that dimension. For a two-dimensional array, they indicate the row and column in which an element is located. Element numbers indicate the sequential position of the element in the array, starting with the first element in the array and increasing in the last dimension first. For a two-dimensional array, the first element is in the first column of the first row, the second element is in the second column of the first row and so on.

To determine the number of dimensions in an array, check its *dimensions* property (it's read-only). The array's *length* property reflects the number of elements in the array. To determine the number of rows or columns in a two-dimensional array, use the *subscript*() method. There is no built-in way to determine the size of dimensions above two.

In an Array object, element numbering starts with zero. You cannot create elements outside the defined range of elements or subscripts (although you could change the dimensions of the array if desired). For example, a 3-row,
4-column array has 12 elements, numbered 0 to 11. The first element's subscripts are [0,0] and the last element is [2,3].

Certain IntraBuilder methods require the element number, and others require the subscripts. If you are using one- or two-dimensional arrays, you can use *element*() to determine the element number if you know the subscripts, and *subscript*() to determine the subscripts if you know the element number.

Array elements may contain any data type, including object references to other arrays. Therefore you can create nested arrays (multi-dimensional arrays of arrays with fixed length in each dimension), ragged arrays (nested arrays with variable lengths), arrays of associative arrays, and so on.

With both nested and multi-dimensional arrays, you end up with multiple dimensions or levels of elements, but when you nest arrays, you create separate array objects, and the methods that are designed to work on the multiple dimensions of a single Array object will not work on the separate dimensions of the nested arrays.

In addition to creating an array with *new*, you can create a populated one-dimensional array using the literal array syntax. For example, this statement

```
a1 = {"A", "B", "C"};
```

creates an Array object with three elements: "A", "B", and "C". You can nest literal arrays. For example, if this statement:

```
a2 = { {1, 2, 3}, a1 };
```

followed the first, you would then have a nested array.

To access a value in a nested array, use the index operators in series. Continuing the example, the third element in the first array would be accessed with:

```
x = a2[0][2];  // 3
```

One-dimensional arrays are the only Array objects that are allowed to have zero elements. This is particularly useful for building arrays dynamically. To create a zero-element array, create a *new* Array with no parameters:

```
a0 = new Array();
```

Then use the *add*() method to add elements to the array.

class Array example

The following statements create a 3 row, 4 column array with the letters “A” through “L” with two different techniques and use a function to display each array.

```
aAlpha = new Array( 3, 4 );
aAlpha[0,0] = "A"; aAlpha[0,1] = "B"; aAlpha[0,2] = "C"; aAlpha[0,3] = "D";
aAlpha[1,0] = "E"; aAlpha[1,1] = "F"; aAlpha[1,2] = "G"; aAlpha[1,3] = "H";
aAlpha[2,0] = "I"; aAlpha[2,1] = "J"; aAlpha[2,2] = "K"; aAlpha[2,3] = "L";
displayArray( aAlpha );
aAlpha = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L" };
aAlpha.resize( 3, 4 );
displayArray( aAlpha );
```

The second array takes advantage of the literal array syntax, but *resize*() only creates a one- or two-dimensional array.

The displayArray() function is used to display the contents of the array in the results pane of the Script Pad. It is shown in the example for *dimensions*.

class AssocArray

Related topics Example

A one-dimensional associative array, in which the elements can be referenced by string.

Syntax

[<oRef> =] new AssocArray()

<oRef>

A variable or property in which to store a reference to the newly created AssocArray object.

Properties

The following tables list the properties and methods of the AssocArray class. (No events are associated with this class.)

Property	Default	Description
className	AssocArray	Identifies the object as an instance of the AssocArray class
firstKey		Character string assigned as the subscript of the first element of an associative array

Method	Parameters	Description
count()		Returns the number of elements in the associative array
isKey()	<key expC>	Returns *true* or *false* to indicate whether the character string is a key of the associative array
nextKey()	<key expC>	Returns the associative array key following the passed key
removeAll()		Deletes all elements from the associative array
removeKey()	<key expC>	Deletes the specified element from the associative array

Description

In an associative array, elements are associated with arbitrary character strings, which act as key values. The keys may be of any length, and are case-sensitive. An AssocArray is a one-dimensional array.

New elements are created simply by assigning a value to a key. If the key does not exist, a new element is created. If the key already exists, then the old value is replaced. For example,

```
aTest = new AssocArray();
aTest[ "alpha" ] = 1  // Create element with key "alpha" value 1
aTest[ "beta"  ] = 2  // Create element with key "beta" value 2
aTest[ "alpha" ] = 3  // Change value of element "alpha" to 3
aTest[ "Beta"  ] = 4  // Create element with key "Beta" value 4
```

The *isKey*() method will check if a given string is a key value in the associative array, and *removeKey*() will remove the element for a given key value from the array. *removeAll*() removes all the elements from the array.

The order of elements in an associative array is undefined. They are not necessarily sorted in the order they were added or sorted by their key values. You can think of an associative array as a bag of elements, and depending on what's in the bag, the order is different. But no matter what's in the associative array, you can use its *firstKey* property to get a key value, and use the *nextKey*() method to get all the other key values. The *count*() method will return the number of elements in the array so that you can call *nextKey*() as many times as needed.

class AssocArray example

Suppose you want to create an associative array that associates country codes with the name of the country. You could use a table for the lookup, but because the lookups don't change, reading the table into an array once at the beginning of the application makes the application run faster.

```
var q = new Query();
q.sql = "select * from COUNTRY";
q.active = true;
var r = q.rowset;
var aCountry = new AssocArray();
while ( !r.endOfSet ) {
  aCountry[ r.fields[ "Code" ].value ] = r.fields[ "Name" ].value;
  r.next();
}
```

If you had to create the array manually, the code would look like this:

```
var aCountry = new AssocArray();
aCountry[ "AFG" ] = "Afghanistan";
aCountry[ "ALB" ] = "Albania";
aCountry[ "ALG" ] = "Algeria";
aCountry[ "ASA" ] = "American Samoa";
Ä
aCountry[ "ZAM" ] = "Zambia";
aCountry[ "ZIM" ] = "Zimbabwe";
```

add()

Related topics Example

Adds an element to a one-dimensional array.

Syntax

<oRef>.*add*(<exp>)

<oRef>

A reference to the one-dimensional array to which you want to add the element.

<exp>

An expression of any type you want to assign to the new element.

Property of

Array

Description

Use *add*() to dynamically build a one-dimensional array.

add() adds a new element to a one-dimensional array and assigns <exp> to the new element.

You can create an empty one-dimensional array in a statement like:

```
a = new Array();  // No parameters to Array class creates empty 1-D array
```

and add elements as needed.

add() example

The following function is an *onServerLoad* event handler for a Select component. It creates a one-dimensional array from values in a field in a table and assigns that array as the *options* property of the Select component. The table is already opened in the query sections1.

```
function sectionSelect_onServerLoad()
{
  this.aSections = new Array();
  this.form.sections1.rowset.first();
  while ( !this.form.sections1.rowset.endOfSet ) {
    this.aSections.add( this.form.sections1.rowset.fields[ "Name" ].value );
    this.form.sections1.rowset.next();
  }
  this.options = "array this.aSections";
}
```

count()

Related topics Example

Returns the number of elements in an associative array.

Property of

AssocArray

Description

Use count() to determine the number of elements in an associative array.

Because associative arrays use arbitrary strings as keys and change size dynamically, you need to get the number of elements in an associative array if you want to loop through its elements.

count() example

The following statements loop through an associative array and display all its elements:

```
var aTest = new AssocArray();
aTest[ "USA" ] = "United States of America";
aTest[ "RUS" ] = "Russian Federation";
aTest[ "GER" ] = "Germany";
aTest[ "CHN" ] = "People's Republic of China";
var cKey = aTest.firstKey;                  // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for ( var nElements = aTest.count(); nElements > 0; nElements-- ) {
  _sys.scriptOut.writeln( cKey );           // Display key value
  _sys.scriptOut.column = 10;               // Line up element value
  _sys.scriptOut.write( aTest[ cKey ] );    // Display element value
cKey = aTest.nextKey( cKey );              // Get next key value
}
```

delete()

Related topics Example

Deletes an element from a one-dimensional array, or deletes a row or column of elements from a two-dimensional array. Returns 1 if successful; generates an error if unsuccessful. The remaining elements move forward to replace the deleted element(s); the dimensions of the array do not change.

Syntax

<oRef>.*delete*(<position expN> [, <row/column expN>])

<oRef>

A reference to the one- or two-dimensional array from which you want to delete data.

<position expN>

When the array is a one-dimensional array, <position expN> specifies the number of the element to delete.

When the array is a two-dimensional array, <position expN> specifies the number of the row or column whose elements you want to delete. The second argument (discussed in the next paragraph) specifies whether <position expN> is a row or a column.

<row/column expN>

Either 1 or 2. If you omit this argument or specify 1, a row is deleted from a two-dimensional array. If you specify 2, a column is deleted. IntraBuilder generates an error if you use <row/column expN> with a one-dimensional array.

Property of

Array

Description

Use *delete*() to delete selected elements from an array without changing the size of the array. *delete*() does the following:

- Deletes an element from a one-dimensional array, or deletes a row or column from a two-dimensional array
- Moves all remaining elements toward the beginning of the array (up if a row is deleted, to the left if an element or column is deleted)
- Inserts *false* values in the last position(s)

Use *resize*() to make the array smaller after you *delete*() if you want the net effect of removing elements.

One-dimensional arrays

When you issue *delete*() for a one-dimensional array, the element in the specified position is deleted, and the remaining elements move one position toward the beginning of the array. The logical value *false* is stored to the element in the last position.

For example, if you define a one-dimensional array with

```
aAlpha = {"A", "B", "C"}
```

the resulting array has one row and can be illustrated as follows:

```
A   B   C
```

Issuing aAlpha.*delete*(1) deletes element number 1 (the second element) whose value is “B,” moves the value in aAlpha[2] to aAlpha[1], and stores *false* to aAlpha[2] so that the array now contains these values:

```
A   C   false
```

Two-dimensional arrays

When you issue *delete*() for a two-dimensional array, the elements in the specified row or column are deleted, and the elements in the remaining rows or columns move one position toward the beginning of the array. The logical value *false* is stored to the elements in the last row or column.

For example, suppose you define a two-dimensional array and store letters to the array. The following

illustration shows how the array is changed by aAlpha.*delete*(1,2).

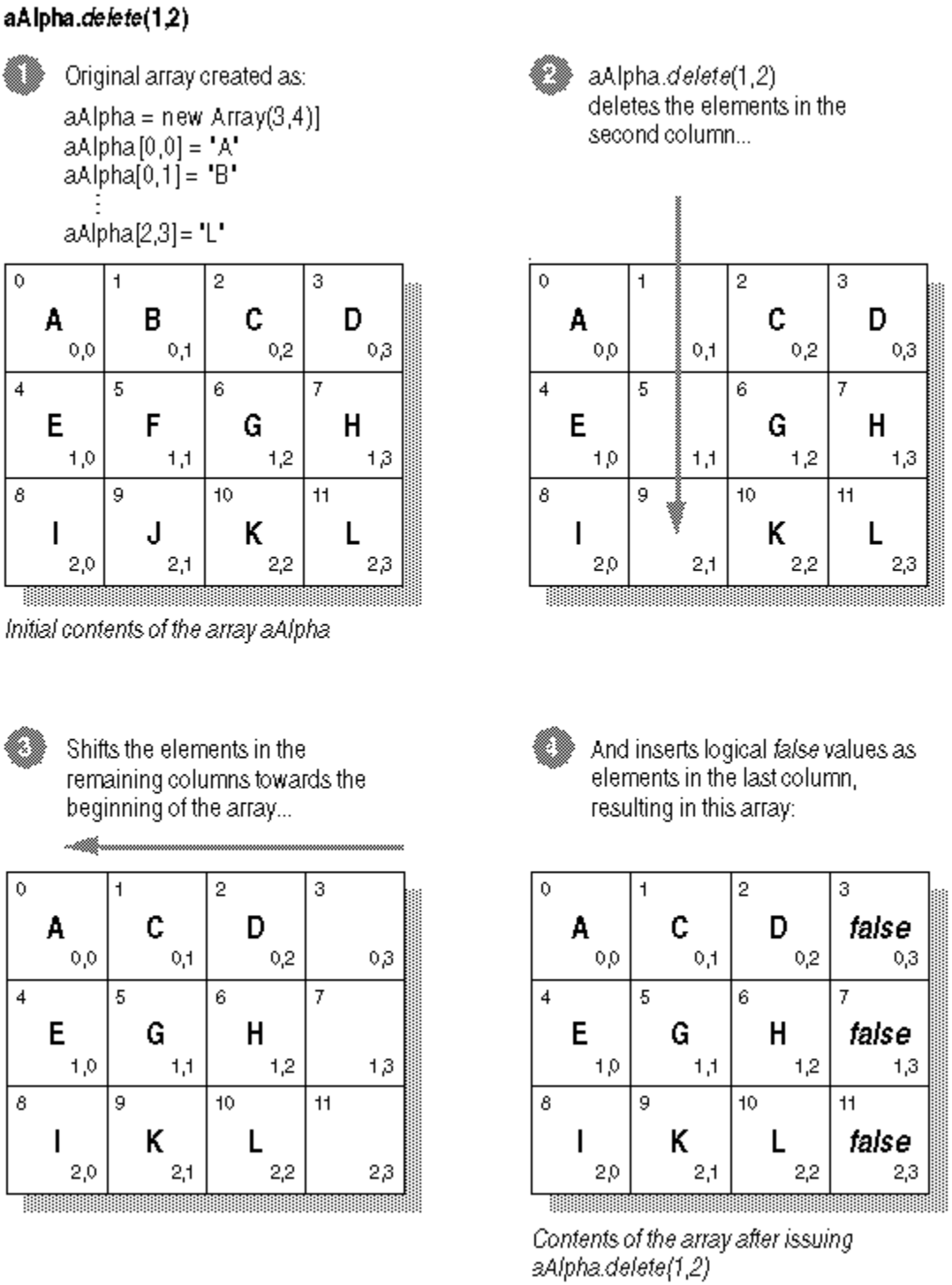


Figure 9.1 Using delete() with a two-dimensional array

delete() example

The following code removes elements from the array aTest that have the letter “e” in them.

```
aTest = {"alpha", "beta", "gamma", "delta"};
var nDeleted = 0;                                   // Count deleted elements
// Loop through array backwards
for ( var nElement = aTest.length - 1; nElement >= 0; nElement-- ) {
  if ( aTest[ nElement ].indexOf( "e" ) >= 0 ) {  // If element contains "e"
    aTest.delete( nElement );                       //   Delete element
    nDeleted++;                                     //   Increment delete count
  }
}
if ( nDeleted > 0 ) {
  if ( nDeleted == aTest.length ) {                 // If all elements deleted
    aTest = new Array();                            //   Recreate empty array
  }
  else {                                            // Otherwise
    aTest.resize( aTest.length - nDeleted );        //   Discard false elements
  }
}
// Display elements (looping forward)
for ( var nElement = 0; nElement < aTest.length; nElement++ ) {
  _sys.scriptOut.writeln( aTest[ nElement ] );
}
```

The loop to delete the elements runs through the array backwards because *delete*() moves all remaining elements forward. You would then have to recheck the same element number and juggle the element counter. It’s simpler to just loop through the array backwards.

After deleting the elements, the array is resized to discard all the *false* elements. If all the elements are deleted, then a new empty array is created, because you cannot *resize*() an array to zero elements.

dimensions

Related topics Example

The number of dimensions in an Array object.

Property of

Array

Description

dimensions indicates the number of dimensions in an Array object. It is a read-only property.

You can use the *resize*() method to change the number of dimensions to one or two, but for more than two you would have to create a new array.

If the array has one or two dimensions, you can use the *subscript*() method to determine the size of each dimension. There is no built-in way to determine dimension sizes for arrays with more than two dimensions.

dimensions example

The following function displays the contents of an array, but only if the array has one or two dimensions:

```
function displayArray( aArg, nColWidth )
{
  #define DEFAULT_WIDTH 2
  if ( displayArray.arguments.length < 2 ) {
    nColWidth = DEFAULT_WIDTH;
  }
  var cLine = new StringEx();
  switch ( aArg.dimensions ) {
    case 1:                                              // 1-D
      _sys.scriptOut.writeln( cLine.replicate( "-", nColWidth * aArg.length )
);
      _sys.scriptOut.writeln();
      for ( var nElement = 0; nElement < aArg.length; nElement++ ) {
        _sys.scriptOut.column = nColWidth * nElement; // Line up columns
        _sys.scriptOut.write( aArg[ nElement ] );     // Display elements
      }                                                  //   in a single line
      break;
    case 2:                                              // 2-D
      var nCols = aArg.subscript( aArg.length - 1, 2 ) + 1 // Determine # of
columns
      var nRows = aArg.length / nCols;                   // Calculate # of rows
      _sys.scriptOut.writeln( cLine.replicate( "-", nColWidth * nCols ) );
      for ( var nRow = 0; nRow < nRows; nRow++ ) {
        _sys.scriptOut.writeln();                        // Each row on its own
line
        for ( var nCol = 0; nCol < nCols; nCol++ ) {  // Display each row as
before
          _sys.scriptOut.column = nColWidth * nCol;
          _sys.scriptOut.write( aArg[ nRow, nCol ] );
        }
      }
      break;
    default:
      alert( "Error: only 1 or 2 dimensions allowed" );
  }
}
```

dir()

Related topics Example

Fills the array with five characteristics of specified files: name, size, modified date, modified time, and DOS attribute(s). Returns the number of files whose characteristics are stored.

Syntax

<oRef>.*dir*([<filename skeleton expC> [, <DOS file attribute list expC>]])

<oRef>

A reference to the array in which you want to store the file information. *dir*() dynamically sizes the array to accommodate the file information.

<filename skeleton expC>

The file-name pattern (using wildcards) describing the files whose information you want to store to <oRef>.

<DOS file attribute list expC>

The letter or letters D, H, and/or S representing one or more DOS file attributes.

If you want to specify a value for <DOS file attribute expC>, you must also specify a value or "*.*" for <filename skeleton expC>.

The meaning of each attribute is as follows:

Character	Meaning
D	Directories
H	Hidden files
S	System files

If you supply more than one letter for <DOS file attribute expC>, include all the letters between one set of quotation marks, for example, aFiles.*dir*("*.*", "HS").

Property of

Array

Description

Use *dir*() to store information about files to an array, which is dynamically resized so all returned information fits in the array. The resulting array is always a two-dimensional array, unless there are no files, in which case the array is not modified.

Without <filename skeleton expC>, *dir*() stores information about all files in the current directory, unless they are hidden or system files. For example, if you want to return information only on DBF tables, use "*.DBF" as <filename skeleton expC>.

If you want to include directories, hidden files, or system files in the array, use <DOS file attribute expC>. When D, H, or S is included in <DOS file attribute expC>, all directories, hidden files, and/or system files (respectively) that match <filename skeleton expC> are added to the array.

dir() stores the following information for each file in each row of the array. The data type for each is shown in parentheses:

Column 0	Column 1	Column 2	Column 3	Column 4
File name (character)	Size (numeric)	Modified date (date)	Modified time (character)	DOS attribute(s) (character)

The last column (DOS attribute) can contain one or more of the following DOS attributes:

Attribute	Meaning
R	Read-only file
A	Archive file (modified since it was last backed up)

S	System file
H	Hidden file
D	Directory

If the file has the attribute, the letter code is in the column. Otherwise, there is a period. For example, a file with none of the attributes would have the following string in column 4:

```
.....
```

A read-only, hidden file would have the following string in column 4:

```
R..H.
```

Use *dirExt*() to get extended Windows 95/NT file information.

dir() example

The following example uses *dir*() to store the file information for all the files in the root directory of the current drive to the array aFiles. The file name and attributes string is displayed for all the files in the results pane of the Script Pad. Manifest constants to represent the columns are created with the #define preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 0  // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 1
#define ARRAY_DIR_DATE 2
#define ARRAY_DIR_TIME 3
#define ARRAY_DIR_ATTR 4
var aFiles = new Array();                  // Array will be resized as needed
var nFiles = aFiles.dir( "\\*.*", "HS" );  // Include Hidden and System files
for ( var nFile = 0; nFile < nFiles; nFile++ ) {
  _sys.scriptOut.writeln( aFiles[ nFile, ARRAY_DIR_NAME ] );
  _sys.scriptOut.column = 25;              // Line up next column
  _sys.scriptOut.write( aFiles[ nFile, ARRAY_DIR_ATTR ] );
}
```

Note the use of double backslashes in the file specification. In a literal string, the backslash acts as an escape character, so you need to use a double backslash for every backslash you want in a literal string.

dirExt()

Related topics Example

dirExt() is an extended version of the *dir*() method. It fills the array with nine characteristics of specified files: name, size, modified date, modified time, DOS attribute(s), short (8.3) file name, create date, create time, and access date. Returns the number of files whose characteristics are stored.

Syntax

<oRef>.*dirExt*([<filename skeleton expC> [, <DOS file attribute list expC>]])

<oRef>

A reference to the array in which you want to store the file information. *dirExt*() dynamically sizes the array to accommodate the file information.

<filename skeleton expC>

The file-name pattern (using wildcards) describing the files whose information you want to store to <oRef>.

<DOS file attribute list expC>

The letter or letters D, H, and/or S representing one or more DOS file attributes.

If you want to specify a value for <DOS file attribute expC>, you must also specify a value or "*.*" for <filename skeleton expC>.

The meaning of each attribute is as follows:

Character	Meaning
D	Directories
H	Hidden files
S	System files

If you supply more than one letter for <DOS file attribute expC>, include all the letters between one set of quotation marks, for example, aFiles.*dirExt*("*.*", "HS").

Property of

Array

Description

Use *dirExt*() to store information about files to an array, which is dynamically resized so all returned information fits in the array. The resulting array is always a two-dimensional array, unless there are no files, in which case the array is not modified.

Without <filename skeleton expC>, *dirExt*() stores information about all files in the current directory, unless they are hidden or system files. For example, if you want to return information only on DBF tables, use "*.DBF" as <filename skeleton expC>.

If you want to include directories, hidden files, or system files in the array, use <DOS file attribute expC>. When D, H, or S is included in <DOS file attribute expC>, all directories, hidden files, and/or system files (respectively) that match <filename skeleton expC> are added to the array.

dirExt() stores the following information for each file in each row of the array. The data type for each is shown in parentheses:

Column 0	Column 1	Column 2	Column 3	Column 4
File name (character)	Size (numeric)	Modified date (date)	Modified time (character)	DOS attribute(s) (character)
Column 5	Column 6	Column 7	Column 8	
Short (8.3) file name (character)	Create date (date)	Create time (character)	Access date (date)	

The column 4 (DOS attribute) can contain one or more of the following DOS attributes:

Attribute	Meaning
R	Read-only file
A	Archive file (modified since it was last backed up)
S	System file
H	Hidden file
D	Directory

If the file has the attribute, the letter code is in the column. Otherwise, there is a period. For example, a file with none of the attributes would have the following string in column 4:

```
.....
```

A read-only, hidden file would have the following string in column 4:

```
R..H.
```

Use *dir*() to get basic file information only.

dirExt() example

The following example uses *dirExt*() to store the file information for all the files in the root directory of the current drive to the array aFiles. The file name and access date is displayed for all the files in the results pane of the Script Pad. Manifest constants to represent the columns are created with the #define preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME        0 // Manifest constants for columns returned by
dirExt()
#define ARRAY_DIR_SIZE        1
#define ARRAY_DIR_DATE        2
#define ARRAY_DIR_TIME        3
#define ARRAY_DIR_ATTR        4
#define ARRAY_DIR_SHORT_NAME  5
#define ARRAY_DIR_CREATE_DATE 6
#define ARRAY_DIR_CREATE_TIME 7
#define ARRAY_DIR_ACCESS_DATE 8
var aFiles = new Array();                   // Array will be resized as
needed
var nFiles = aFiles.dirExt( "\\*.*", "HS" ); // Include Hidden and System
files
for ( var nFile = 0; nFile < nFiles; nFile++ ) {
  _sys.scriptOut.writeln( aFiles[ nFile, ARRAY_DIR_NAME ] );
  _sys.scriptOut.column = 25;               // Line up next column
  _sys.scriptOut.write( aFiles[ nFile, ARRAY_DIR_ACCESS_DATE ] );
}
```

Note the use of double backslashes in the file specification. In a literal string, the backslash acts as an escape character, so you need to use a double backslash for every backslash you want in a literal string.

element()

Related topics Example

Returns the number of a specified element in a one- or two-dimensional array.

Syntax

<oRef>.*element*(<subscript1 expN> [, <subscript2 expN>])

<oRef>

A reference to a one- or two-dimensional array.

<subscript1 expN>

The first subscript of the element. In a one-dimensional array, this is the same as the element number. In a two-dimensional array, this is the row.

<subscript2 expN>

When <oRef> is a two-dimensional array, <subscript2 expN> specifies the second subscript, or column, of the element.

If <oRef> is a two-dimensional array and you do not specify a value for <subscript2 expN>, IntraBuilder assumes the value 0, the first column in the row. IntraBuilder generates an error if you use <subscript2 expN> with a one-dimensional array.

Property of

Array

Description

Use *element*() when you know the subscripts of an element in a two-dimensional array and need the element number for use with another method, such as *fill*() or *scan*().

In one-dimensional arrays, the number of an element is the same as its subscript, so there is no need to use *element*(). For example, if aOne is a one-dimensional array, aOne.*element*(3) returns 3, aOne.*element*(5) returns 5, and so on.

element() is the inverse of *subscript*(), which returns an element's row or column subscript number when you specify the element number.

element() example

The following statement returns the element number of the third column of the fourth row of the array aTwo. The result depends on the number of columns in aTwo.

```
nElement = aTwo.element( 3, 2 ); // Fourth row, third column
```

fill()

Related topics Example

Stores a specified value into one or more locations in an array, and returns the number of elements stored.

Syntax

<oRef>.*fill*(<exp> [, <start expN> [, <count expN>]])

<oRef>

A reference to a one- or two-dimensional array you want to fill with the specified value <exp>.

<exp>

An expression you want to store in the specified array.

<start expN>

The element number at which you want to begin storing <exp>.
If you do not specify <start expN>, IntraBuilder begins at the first element in the array.

<count expN>

The number of elements in which you want to store <exp>, starting at element <start expN>. If you do not specify <count expN>, IntraBuilder stores <exp> from <start expN> to the last element in the array. If you want to specify a value for <count expN>, you must also specify a value for <start expN>.

If you do not specify <start expN> or <count expN>, IntraBuilder fills all elements in the array with <exp>.

Property of

Array

Description

Use *fill*() to store a value into all or some elements of an array. For example, if you are going to use elements of an array to calculate totals, you can use *fill*() to initialize all values in the array to 0.

fill() stores values into the array sequentially. Starting at the first element in the array or at the element specified by <start expN>, *fill*() stores the value in each element in a row, then moves to the first element in the next row, continuing to store values until the array is filled or until it has inserted <count expN> elements. *fill*() overwrites any existing data in the array.

If you know an element's subscripts, you can use *element*() to determine its element number for use as <start expN>.

fill() example

Suppose you're measuring the performance of a process, keeping track of six different variables, some of which may not used for any given request. In addition to keeping an average, you want to always display the last three measurements. You can use an array with 3 rows and 6 columns, and *insert*() a new row at the beginning of the array for each request. You *fill*() the new row with zeros to initialize the variables in case they're not used. The code, with simulated input, would look like this:

```
#define SHOW_LAST        3  // Manifest constants for number of measurements
#define NUM_MEASUREMENTS 6  //   to maintain
aMeasure = new Array( SHOW_LAST, NUM_MEASUREMENTS );
aMeasure.fill( "" );          // Start with all blanks
// Simulated input
newRequest();
aMeasure[ 0, 1 ] = 34;
aMeasure[ 0, 4 ] = 16;
displayArray( aMeasure, 10 );
newRequest();
aMeasure[ 0, 3 ] = 67;
displayArray( aMeasure, 10 );
newRequest();
aMeasure[ 0, 0 ] = 27;
aMeasure[ 0, 1 ] = 29;
displayArray( aMeasure, 10 );
newRequest();
aMeasure[ 0, 1 ] = 31;
aMeasure[ 0, 5 ] = 40;
displayArray( aMeasure, 10 );
// End simulated input
function newRequest()
{
  aMeasure.insert( 0 );                    // Insert row at top, losing last
row
  aMeasure.fill( 0, 0, NUM_MEASUREMENTS );  // Fill first row with zeros
}
```

The displayArray() function is used to display the contents of the array in the results pane of the Script Pad. It is shown in the example for *dimensions*.

firstKey

Related topics Example

Returns the character string key for the first element of an associative array.

Property of

AssocArray

Description

Use *firstKey* when you want to loop through the elements in an associative array. Once you have gotten the key value for the first element with *firstKey*, use *nextKey*() to get the key values for the rest of the elements.

Note The order of elements in an associative array is undefined. They are not necessarily stored in the order in which you add them, or sorted by their key values. You can't assume that the value returned by *firstKey* will be consistent, or that it will return the first item you added.

For an empty associative array, *firstKey* is the logical value *false*. Because *false* is a different data type than valid key values (which are character strings), it's difficult to look for *false* to see if the array is empty. It's easier to get the number of elements in the array with *count*() and see if it's greater than zero.

firstKey example

The following statements loop through an associative array and display all its elements:

```
var aTest = new AssocArray();
aTest[ "USA" ] = "United States of America";
aTest[ "RUS" ] = "Russian Federation";
aTest[ "GER" ] = "Germany";
aTest[ "CHN" ] = "People's Republic of China";
var cKey = aTest.firstKey;                  // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for ( var nElements = aTest.count(); nElements > 0; nElements-- ) {
  _sys.scriptOut.writeln( cKey );           // Display key value
  _sys.scriptOut.column = 10;               // Line up element value
  _sys.scriptOut.write( aTest[ cKey ] );    // Display element value
cKey = aTest.nextKey( cKey );             // Get next key value
}
```

grow()

Related topics Example

Adds an element, row, or column to an array and returns the number of added elements.

Syntax

<oRef>.*grow*(<expN>)

<oRef>

A reference to a one- or two-dimensional array you want to add elements to.

<expN>

Either 1 or 2. When you specify 1, *grow*() adds a single element to a one-dimensional array or a row to a two-dimensional array. When you specify 2, *grow*() adds a column to the array.

Property of

Array

Description

Use *grow*() to insert an element, row, or column into an array and change the size of the array to reflect the added elements. *grow*() can make a one-dimensional array two-dimensional. All added elements are initialized to *false* values.

One-dimensional arrays

When you specify 1 for <expN>, *grow*() adds a single element to the array. When you specify 2, *grow*() makes the array two-dimensional, and existing elements are moved into the first column. This is shown in the following figure:

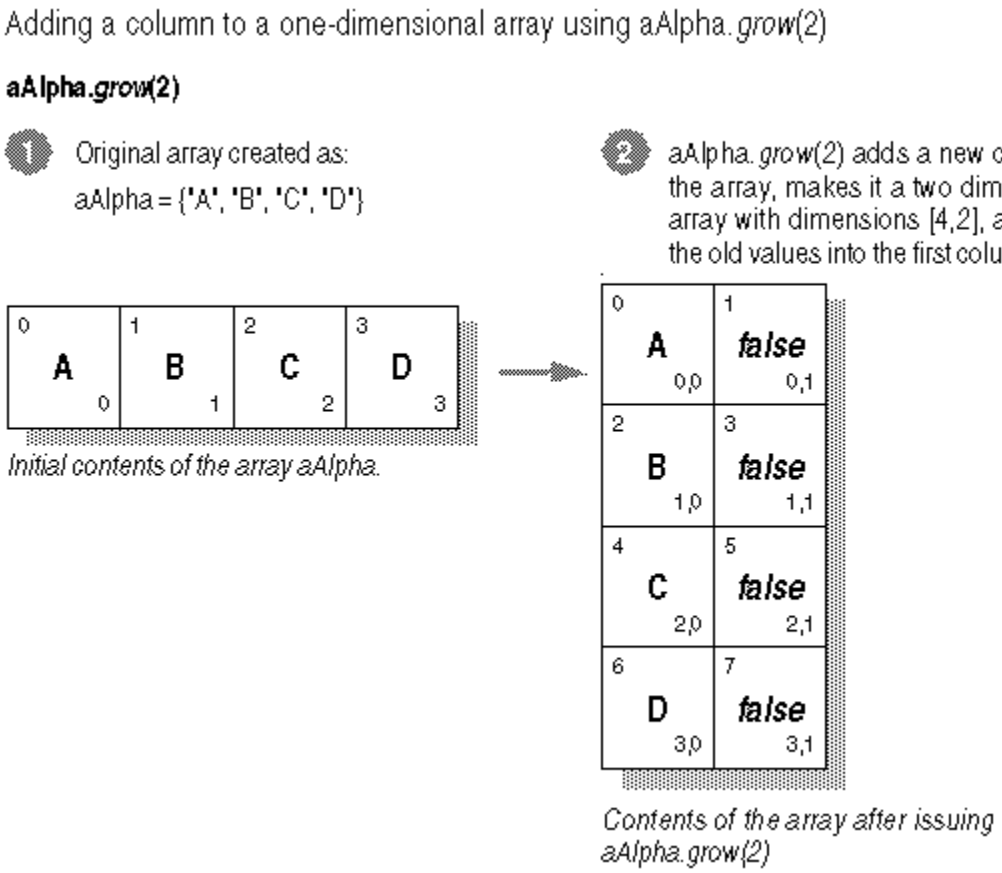


Figure 9.2 Adding a column to a one-dimensional array using aAlpha.*grow*(2)

Use *add*() to add a new element to a one-dimensional array and assign its value in one step.

Two-dimensional arrays

When you specify 1 for <expN>, *grow*() adds a row to the array at the end of the array. This is shown in the following figure:

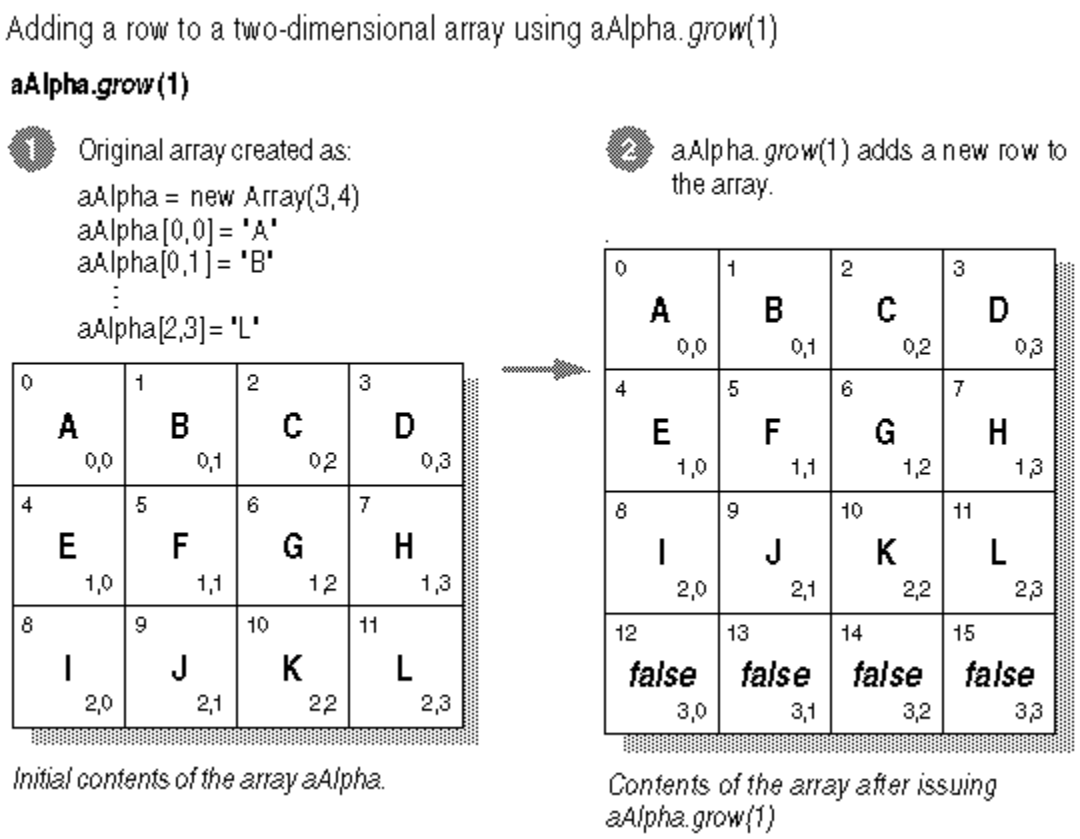


Figure 9.3 Adding a row to a two-dimensional array using aAlpha.*grow*(1)

When you specify 2 for <expN>, *grow*() adds a column to the array and places *false* into each element in the column.

grow() example

The following example initially declares a one-dimensional array with a single element, and then uses *grow*() to add a second element, convert the array to two dimensions, add a third row, and finally add a third column. The end result is the first nine letters in order:

```
a = {"A"};          // 1-D, 1 element
displayArray( a );
a.grow(1);          // 1-D, 2 elements
a[ 1 ] = "D";
displayArray( a );
a.grow(2);          // 2-D, 2 rows, 2 columns
a[ 0, 1 ] = "B"; a[ 1, 1 ] = "E";
displayArray( a );
a.grow(1);          // 2-D, 3 rows, 2 columns
a[ 2, 0 ] = "G"; a[ 2, 1 ] = "H";
displayArray( a );
a.grow(2);          // 2-D, 3 rows, 3 columns
a[ 0, 2 ] = "C"; a[ 1, 2 ] = "F"; a[ 2, 2 ] = "I";
displayArray( a );
```

The displayArray() function is used to display the contents of the array in the results pane of the Script Pad. It is shown in the example for ***dimensions***.

insert()

Related topics Example

Inserts an element with the value *false* into a one-dimensional array, or inserts a row or column of elements with the value *false* into a two-dimensional array. Returns 1 if successful; generates an error if unsuccessful. The dimensions of the array do not change, so the element(s) at the end of the array will be lost.

Syntax

<oRef>.*insert*(<position expN> [, <row/column expN>])

<oRef>

A reference to a one- or two-dimensional array in which you want to insert data.

<position expN>

When <oRef> is a one-dimensional array, <position expN> specifies the number of the element in which you want to insert a *false* value.

When <oRef> is a two-dimensional array, <position expN> specifies the number of a row or column in which you want to insert *false* values. The second argument (discussed in the next paragraph) specifies whether <position expN> is a row or a column.

<row/column expN>

Either 1 or 2. If you omit this argument or specify 1, a row is inserted into a two-dimensional array. If you specify 2, a column is inserted. IntraBuilder generates an error if you use <row/column expN> with a one-dimensional array.

Property of

Array

Description

Use *insert*() to insert elements in an array. *insert*() does the following:

- Inserts an element in a one-dimensional array, or inserts a row or column in a two-dimensional array
- Moves all remaining elements toward the end of the array (down if a row is inserted, to the right if an element or column is inserted)
- Stores *false* values in the newly created position(s)

Because the dimensions of the array are not changed, the element(s) at the end of the array—the last element for a one-dimensional array or the last row or column for a two-dimensional array—are lost. If you don't want to lose the data, use *grow*() to increase the size of the array before using *insert*().

One-dimensional arrays

When you call *insert*() for a one-dimensional array, the logical value *false* is inserted into the position of the specified element. The remaining element(s) are moved one place toward the end of the array. The element that had been in the last position is lost.

For example, if you define a one-dimensional array with:

```
aAlpha = {"A", "B", "C"}
```

the resulting array has one row and can be illustrated as follows:

```
A   B   C
```

Issuing aAlpha.*insert*(1) inserts *false* into element number 1 (the second element), moves the "B" that was in aAlpha[1] to aAlpha[2], and loses the "C" that was in aAlpha[2] so that the array now contains these values:

```
A  false  B
```

Two-dimensional arrays

When you call *insert*() for a two-dimensional array, a logical value *false* is inserted into the position of each element in the specified row or column. The elements in the remaining columns or rows are moved one place toward the end of the array. The elements that had been in the last row or column are lost.

For example, suppose you define a two-dimensional array and store letters to the array. The following illustration shows how the array is changed by aAlpha.*insert*(1,2).

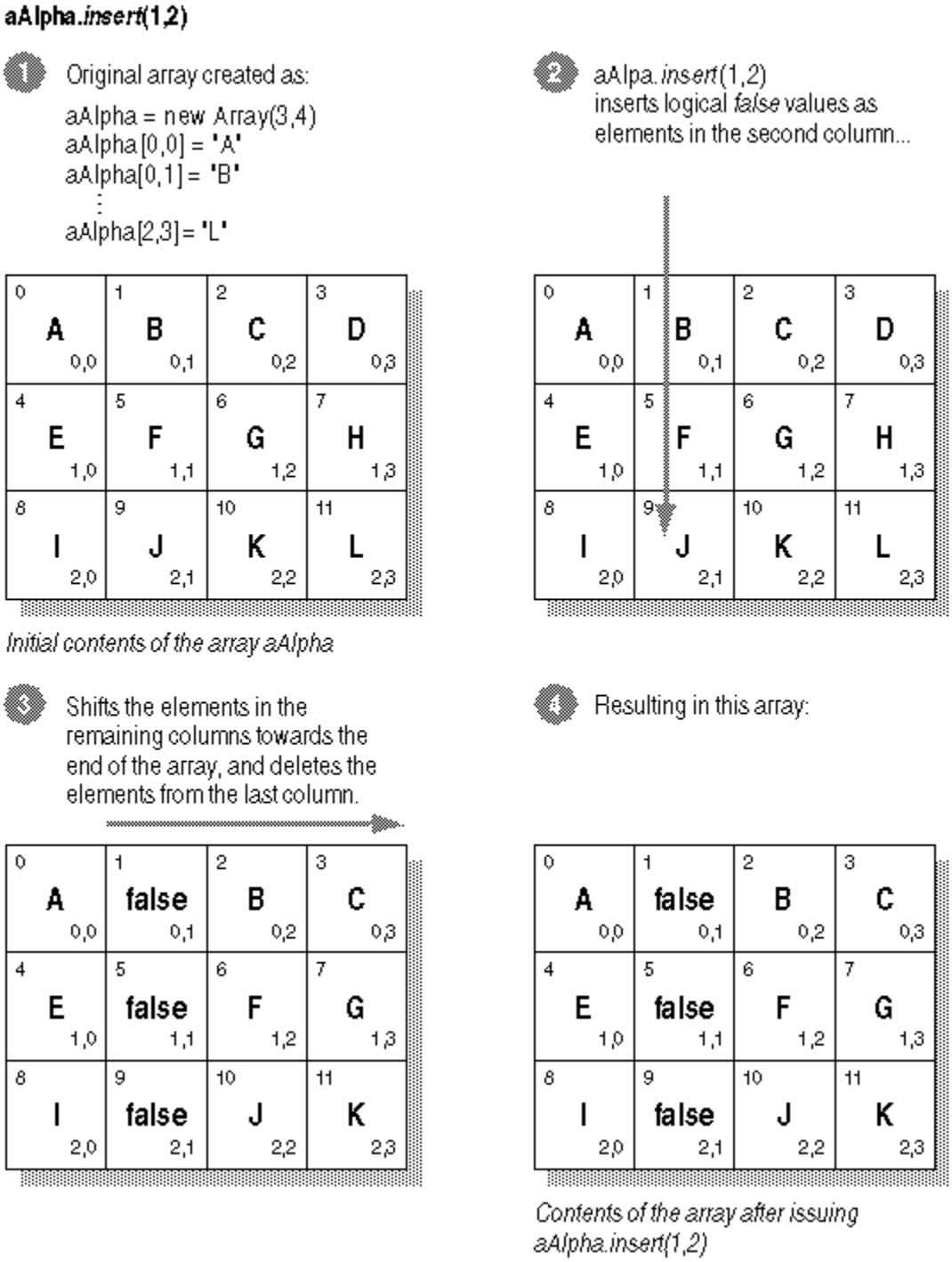


Figure 9.4 Using insert() with a two-dimensional array

insert() example

Suppose you're measuring the performance of a process, keeping track of six different variables, some of which may not used for any given request. In addition to keeping an average, you want to always display the last three measurements. You can use an array with 3 rows and 6 columns, and *insert*() a new row at the beginning of the array for each request. You *fill*() the new row with zeros to initialize the variables in case they're not used. The code, with simulated input, would look like:

```
#define SHOW_LAST        3  // Manifest constants for number of measurements
#define NUM_MEASUREMENTS 6  //   to maintain
aMeasure = new Array( SHOW_LAST, NUM_MEASUREMENTS );
aMeasure.fill( "" );          // Start with all blanks
// Simulated input
newRequest();
aMeasure[ 0, 1 ] = 34;
aMeasure[ 0, 4 ] = 16;
displayArray( aMeasure, 10 );
newRequest();
aMeasure[ 0, 3 ] = 67;
displayArray( aMeasure, 10 );
newRequest();
aMeasure[ 0, 0 ] = 27;
aMeasure[ 0, 1 ] = 29;
displayArray( aMeasure, 10 );
newRequest();
aMeasure[ 0, 1 ] = 31;
aMeasure[ 0, 5 ] = 40;
displayArray( aMeasure, 10 );
// End simulated input
function newRequest()
{
  aMeasure.insert( 0 );                    // Insert row at top, losing last
row
  aMeasure.fill( 0, 0, NUM_MEASUREMENTS );  // Fill first row with zeros
}
```

The displayArray() function is used to display the contents of the array in the results pane of the Script Pad. It is shown in the example for ***dimensions***.

isKey()

Related topics Example

Returns a logical value that indicates if the specified character expression is the key of an element in an associative array.

Syntax

<oRef>.*isKey*(<expC>)

<oRef>

A reference to the associative array you want to search.

<expC>

The character string you want to find.

Property of

AssocArray

Description

Use *isKey*(<expC>) to determine if an associative array contains an element with a key value of <expC>. Key values in associative arrays are case-sensitive.

Attempting to access a non-existent key value in an associative array generates an error.

isKey() example

The following example uses some test data for the associative array aCountry, that associates country codes with their names. The function countryName() returns the corresponding country name for a particular code, but if the code is not defined, it returns “Unknown country” instead.

```
var aCountry = new AssocArray();
aCountry[ "USA" ] = "United States of America";  // Test data
aCountry[ "RUS" ] = "Russian Federation";
aCountry[ "GER" ] = "Germany";
aCountry[ "CHN" ] = "People's Republic of China";
_sys.scriptOut.writeln( countryName( "GER" ) );  // "Germany"
_sys.scriptOut.writeln( countryName( "XYZ" ) );  // "Unknown country"
function countryName( cArg ) {
  // Make sure code is defined before trying to reference it
  return aCountry.isKey( cArg ) ? aCountry[ cArg ] : "Unknown country"
}
```

length

Related topics Example

The number of elements in an Array object.

Property of

Array

Description

length indicates the number of elements in an Array object.

For a one-dimensional array, you can assign a value to *length* to change its size.

For a array with more than one dimension, *length* is read-only.

You can use the *subscript*() method to determine the size of each dimension for a two-dimensional array. There is no built-in way to determine dimension sizes for arrays with more than two dimensions.

length example

Whenever you call a function or method, an *arguments* array is created. The *length* property of the *arguments* array indicates the number of parameters passed to the function or method. You can use this value to set default values for parameters that are not passed.

For example, the displayArray() function accepts two parameters: an object reference to an array to display, and a column width. If the column width is not specified, it is set to a default width.

```
function displayArray( aArg, nColWidth )
{
  #define DEFAULT_WIDTH 2
  if ( displayArray.arguments.length < 2 ) {
    nColWidth = DEFAULT_WIDTH;
  }
  // Rest of function....
}
```

nextKey()

Related topics Example

Returns the key value of the element following the specified key in an associative array.

Syntax

<oRef>.*nextKey*(<key expC>)

<oRef>

A reference to the associative array that contains the key.

<key expC>

An existing key value.

Property of

AssocArray

Description

Use *nextKey*() to loop through the elements in an associative array. Once you have gotten the key value for the first element with *firstKey*, use *nextKey*() to get the key values for the rest of the elements.

nextKey() returns the key value for the key following <key expC>. Key values in associative arrays are case-sensitive. For the last key in the associative array and for a <key expC> that is not an existing key value, *nextKey*() returns the logical value *false*. Because *false* is a different data type than valid key values (which are character strings), it's difficult to look for *false* to terminate a loop. It's easier to get the number of elements in the array first with *count*(); then loop through that many iterations.

Note The order of elements in an associative array is undefined. They are not necessarily stored in the order in which you add them, or sorted by their key values. You can't assume that the sequence of keys will be consistent.

To determine if a given character string is a key value in an associative array, use *isKey*().

nextKey() example

The following statements loop through an associative array and display all its elements:

```
var aTest = new AssocArray();
aTest[ "USA" ] = "United States of America";
aTest[ "RUS" ] = "Russian Federation";
aTest[ "GER" ] = "Germany";
aTest[ "CHN" ] = "People's Republic of China";
var cKey = aTest.firstKey;                  // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for ( var nElements = aTest.count(); nElements > 0; nElements-- ) {
  _sys.scriptOut.writeln( cKey );           // Display key value
  _sys.scriptOut.column = 10;               // Line up element value
  _sys.scriptOut.write( aTest[ cKey ] );    // Display element value
cKey = aTest.nextKey( cKey );             // Get next key value
}
```

removeAll()

Related topics Example

Deletes all elements from an associative array.

Syntax

<oRef>.*removeAll*()

<oRef>

A reference to the associative array you want to empty.

Property of

AssocArray

Description

Use *removeAll*() to remove all the elements from an associative array.

To remove elements for particular key values, use *removeKey*().

removeAll() example

The following example removes all elements from an associative array.

```
var aTest = new AssocArray();
aTest[ "USA" ] = "United States of America";
aTest[ "RUS" ] = "Russian Federation";
aTest[ "GER" ] = "Germany";
aTest[ "CHN" ] = "People's Republic of China";
// Array contains four elements
aTest.removeAll(); // Array now contains no elements
```

removeKey()

Related topics Example

Deletes an element from an associative array.

Syntax

<oRef>.*removeKey*(<key expC>)

<oRef>

A reference to the associative array that contains the key.

<key expC>

The key value of the element you want to delete.

Property of

AssocArray

Description

Use *removeKey*() to remove an element from an associative array. Key values in associative arrays are case-sensitive.

If you specify a key value that does not exist in the array, nothing happens; no error occurs and no elements are removed.

To remove all the elements from an associative array, use *removeAll*().

removeKey() example

The following example loops through an associative array of country names and deletes those whose names are longer than 15 characters.

```
var aTest = new AssocArray();
aTest[ "USA" ] = "United States of America";
aTest[ "RUS" ] = "Russian Federation";
aTest[ "GER" ] = "Germany";
aTest[ "CHN" ] = "People's Republic of China";
var cKey = aTest.firstKey;              // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for ( var nElements = aTest.count(); nElements > 0; nElements-- ) {
  var cNextKey = aTest.nextKey( cKey ); // Get next key value before deleting
element
  if ( aTest[ cKey ].length > 15 ) {
    aTest.removeKey( cKey );            // Remove element
  }
  cKey = cNextKey;                      // Use next key value
}
```

Note that you must get the next key value before deleting the element, and you repeat the loop based on the number of elements there were before you started deleting.

resize()

Related topics Example

Sets the size of an array to the specified dimensions and returns a numeric value representing the number of elements in the modified array.

Syntax

<oRef>.*resize*(<rows expN> [,<cols expN> [, <retain values expN>]])

<oRef>

A reference to the array whose size you want to change.

<rows expN>

The number of rows the resized array should have. <rows expN> must always be a positive, nonzero value.

<cols expN>

The number of columns the resized array should have. <cols expN> must always be 0 or a positive value. If you omit this option, *resize*() changes the number of rows in the array and leaves the number of columns the same.

<retain values expN>

Determines what happens to the values of the array when rows are added or removed. If it is nonzero, values are retained. If you want to specify a value for <retain values expN>, you must also specify a value for <new cols expN>.

Property of

Array

Description

Use *resize*() to change the dimensions of an array, making it larger or smaller, or change the number of dimensions. To determine the number of dimensions, check the array's *dimensions* property. The *length* property of the array reflects the number of elements; for a one-dimensional array, that's all you need to know. For a two-dimensional array, you can't determine the number of rows or columns from the *length* property alone (unless the *length* is one—a one-by-one array).

To determine the number of columns in a two-dimensional array, use the *subscript*() method to get the column subscript of the last element in the array, then add one, since subscripts are zero-based. For example,

```
nCol = aExample.subscript( aExample.length - 1, 2 ) + 1;
```

You can use the same technique to get the number of rows, specifying 1 instead of 2 as the second parameter. Since you know the number of columns, you can also calculate the number of rows by dividing the *length* of the array by the number of columns.

For a one-dimensional array, you can change the number of elements by calling *resize*() and specifying the number of elements as <rows expN> parameter. You can also set the *length* property of the array directly, which is a bit less typing.

You can also change a one-dimensional array into a two-dimensional array by specifying both a <rows expN> and a nonzero <cols expN> parameter. This makes the array the designated size.

For a two-dimensional array, you can specify a new number of rows or both row and column dimensions for the array. If you omit <cols expN>, the <rows expN> parameter sets the number of rows only. With both a <rows expN> and a nonzero <cols expN>, the array is changed to the designated size.

You can change a two-dimensional array to a one-dimensional array by specifying <cols expN> as zero and <rows expN> as the number of elements.

To change the number of columns only for a two-dimensional array, you will need to specify both the <rows expN> and <cols expN> parameters, which means that you have to determine the number of rows in the array, if not known, and specify it unchanged as the <rows expN> parameter.

To add a single row or column to an array, use the *grow*() method.

If you add or remove columns from the array, you can use <retain values expN> to specify how you want existing elements to be placed in the new array. If <retain values *expN*> is zero or isn't specified, resize() rearranges the elements, filling in the new rows or columns or adjusting for deleted elements, and adding or removing elements at the end of the array, as needed. This is shown in the following two figures. You are most likely to want to do this if you don't need to refer to existing items in the array; that is, you plan to update the array with new values.

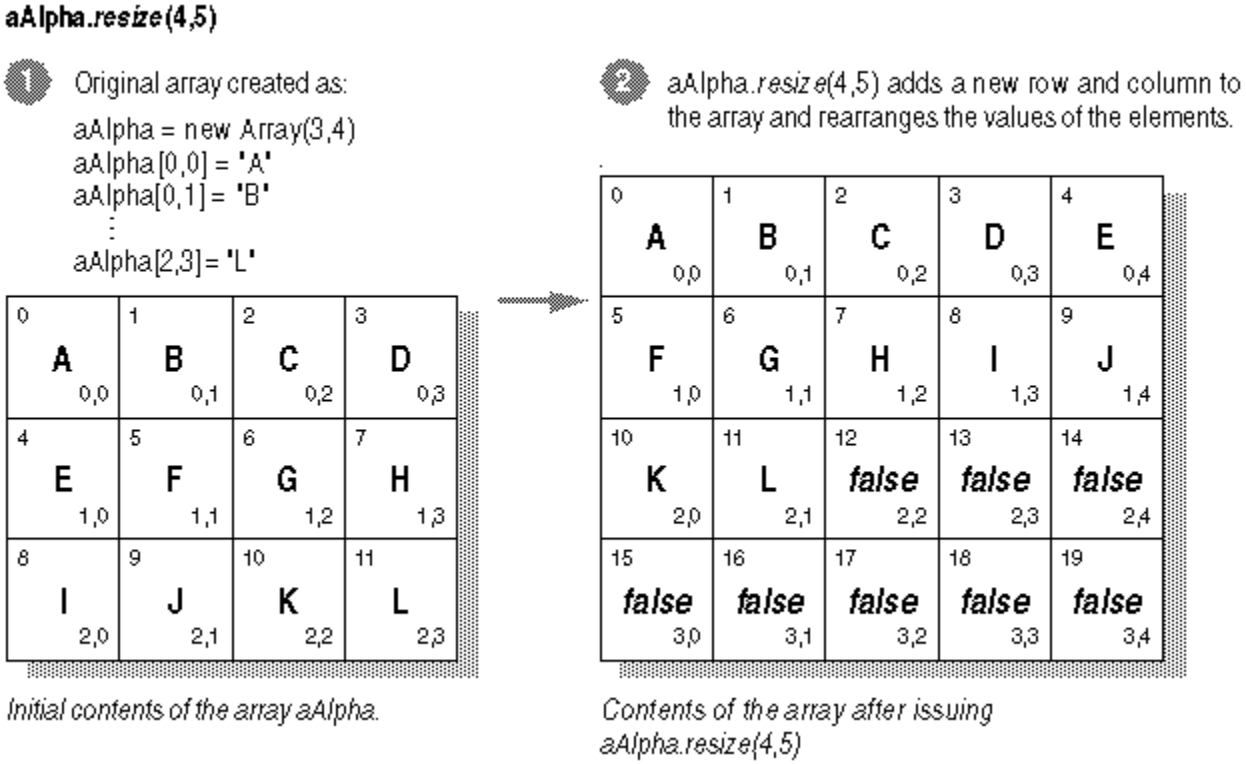


Figure 9.5 Adding a row and a column to a 3x4 array, rearranging elements

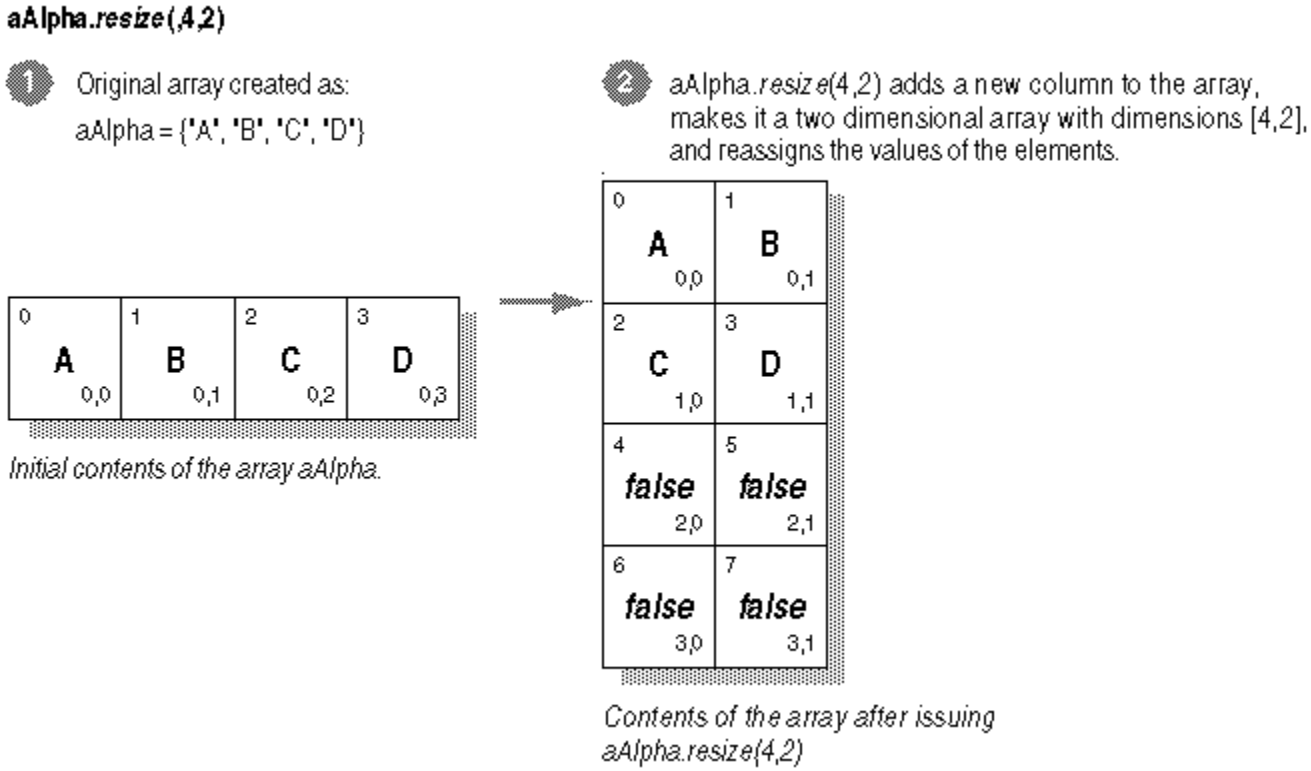


Figure 9.6 Adding a column to a one-dimensional array, rearranging elements

When you use *resize*() on a one-dimensional array, you might want the original row to become the first column of the new array. Similarly, when you use *resize*() on a two-dimensional array, you might want existing two-dimensional array elements to remain in their original positions. You are most likely to want to do this if you need to refer to existing items in the array by their subscripts; that is, you plan to add new values to the array while continuing to work with existing values.

If <retain values expN> is a nonzero value, *resize*() ensures that elements retain their original values. The following two figures repeat the statements shown in the previous two figures, with the addition of a value of 1 for <retain values expN>.

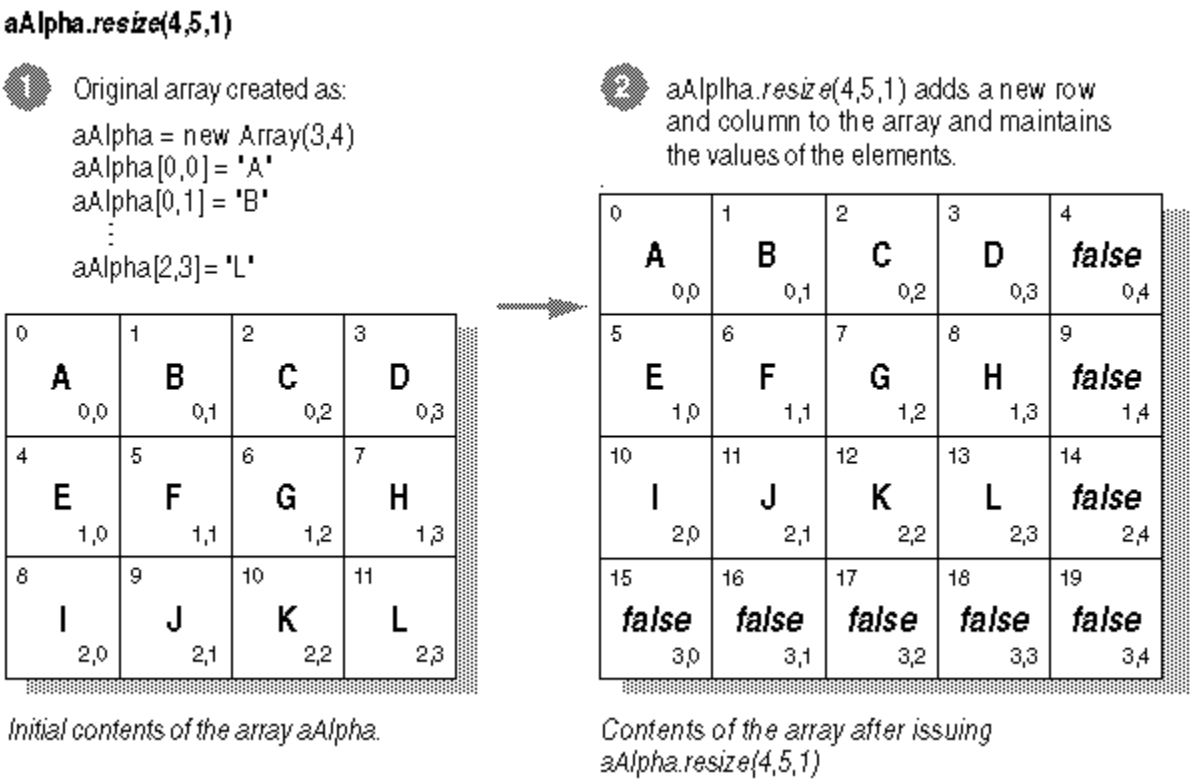


Figure 9.7 Adding a row and a column to a 3x4 array, "preserving elements"

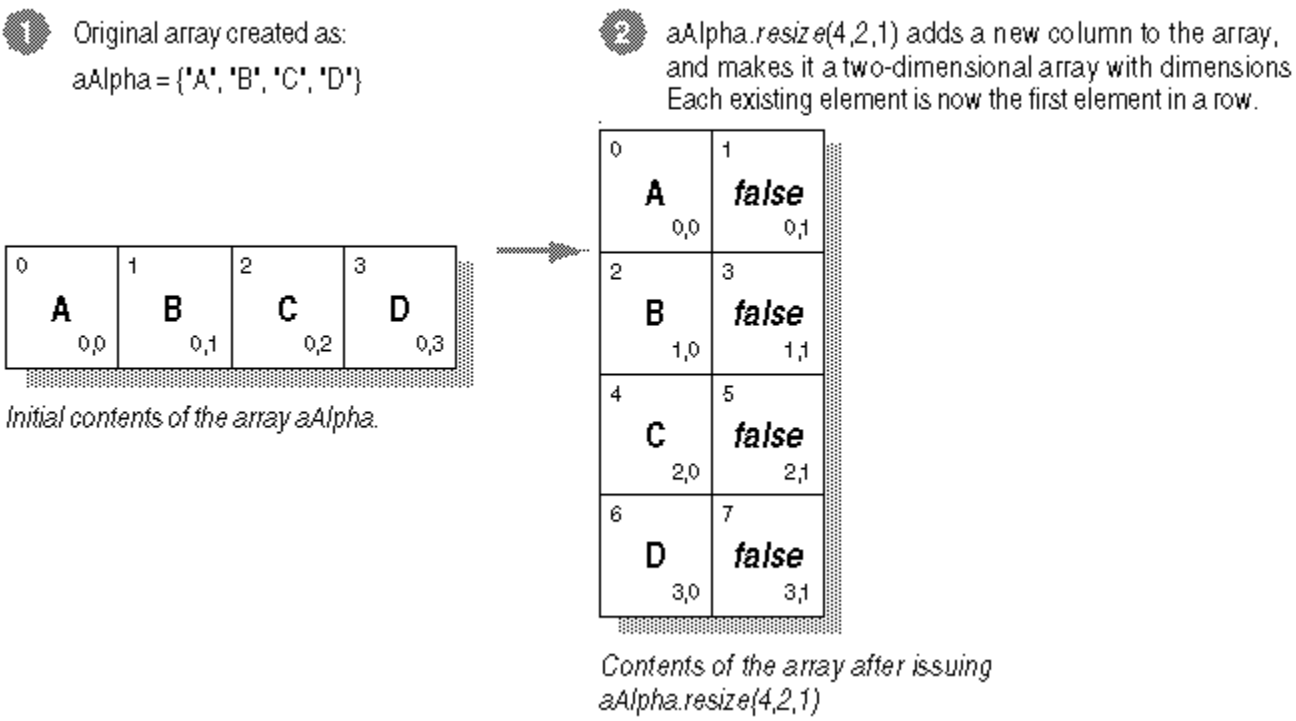


Figure 9.8 Adding a column to a one-dimensional array, "preserving elements"

resize() example

The following code removes elements from the array aTest that have the letter “e” in them.

```
aTest = {"alpha", "beta", "gamma", "delta"};
var nDeleted = 0;                                   // Count deleted elements
// Loop through array backwards
for ( var nElement = aTest.length - 1; nElement >= 0; nElement-- ) {
  if ( aTest[ nElement ].indexOf( "e" ) >= 0 ) {  // If element contains "e"
    aTest.delete( nElement );                       //   Delete element
    nDeleted++;                                     //   Increment delete count
  }
}
if ( nDeleted > 0 ) {
  if ( nDeleted == aTest.length ) {                 // If all elements deleted
    aTest = new Array();                            //   Recreate empty array
  }
  else {                                            // Otherwise
    aTest.resize( aTest.length - nDeleted );        //   Discard false elements
  }
}
// Display elements (looping forward)
for ( var nElement = 0; nElement < aTest.length; nElement++ ) {
  _sys.scriptOut.writeln( aTest[ nElement ] );
}
```

The loop to delete the elements runs through the array backwards because *delete*() moves all remaining elements forward. You would then have to recheck the same element number and juggle the element counter. It’s simpler to just loop through the array backwards.

After deleting the elements, the array is resized to discard all the *false* elements. If all the elements are deleted, then a new empty array is created, because you cannot *resize*() an array to zero elements.

scan()

Related topics Example

Searches an array for an expression. Returns the number of the first element that matches the expression if the search is successful, or –1 if the search is unsuccessful.

Syntax

<oRef>.*scan*(<exp> [, <starting element expN> [, <elements expN>]])

<oRef>

A reference to the array you want to search.

<exp>

The expression you want to search for in <oRef>.

<starting element expN>

The element number in <oRef> at which you want to start searching. Without <starting element expN>, *scan*() starts searching at the first element.

<elements expN>

The number of elements in <oRef> that *scan*() searches. Without <elements expN>, *scan*() searches <oRef> from <starting element expN> to the end of the array. If you want to specify a value for <elements expN>, you must also specify a value for <starting element expN>.

Property of

Array

Description

Use *scan*() to search an array for the value of <exp>. For example, if an array contains customer names, you can use *scan*() to find the location in which a particular name appears.

scan() returns the element number of the first element in the array that matches <exp>. If you want to determine the subscripts of this element, use *subscript*().

When <exp> is a string, *scan*() is case-sensitive; you may want to use the strings's *toUpperCase*(), *toLowerCase*(), or *toProperCase*() methods to match the case of <exp> with the case of the data stored in the array.

scan() example

The following example uses *dir*() to store the file information for all the files and directories in the root directory of the current drive to the array aFiles. Then the array is searched to display only the directories in the array. Manifest constants to represent the columns are created with the #define preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 0  // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 1
#define ARRAY_DIR_DATE 2
#define ARRAY_DIR_TIME 3
#define ARRAY_DIR_ATTR 4
var aFiles = new Array();
var nFiles = aFiles.dir( "\\*.*", "D" );        // Read all files and
directories
var nElement = 0;                               // Start looking at first
element
while ( nElement >= 0 ) {                       // Until there's no match
  nElement = aFiles.scan( "....D", nElement );  // Look for next directory
  if ( nElement >= 0 ) {                        // Display a match
    _sys.scriptOut.writeln( aFiles[ nElement - ARRAY_DIR_ATTR +
ARRAY_DIR_NAME ] );
    if ( ++nElement >= aFiles.length ) {        // Continue looking with next
element
      break;                                    // Unless that was the last
element
    }
  }
}
```

To find all the matches in the array, you need to keep track of the last match. Here it's kept in the variable nElement. It starts at zero, the first element, and is used in the *scan*() call as the starting element parameter. The result of each search is stored back in nElement. If there's a match, the directory name is displayed. Then nElement is incremented—otherwise *scan*() would match the same element again—and the loop continues.

A few subtleties are present in the example code. First, after incrementing nElement, it is compared with the *length* of the array. If the element number is equal to (or greater than, which it should never be, but it's good defensive programming to test for it anyway) the *length* of the array, that means the last match was in the last element of the array. This is possible only because the file attribute is in the last column of the array. In this case, you don't want to call *scan*() again, since the starting element number is higher than the highest element number and would cause an error. So you *break* out of the loop instead.

The variable nElement is incremented before the comparison to the *length* of the array by using the prefix ++ operator. If nElement was post-incremented, the comparison would be off, although the rest of the loop would work.

To display the directory name, the column number of the file attribute column is subtracted from the matching element number, and the column number for the file name column is added. This yields the element number of the file name in the same row as the matching file attribute. This would work for any combination of search or display columns.

sort()

Related topics Example

Sorts the elements in a one-dimensional array or the rows in a two-dimensional array. Returns 1 if successful; generates an error if unsuccessful.

Syntax

<oRef>.*sort*([<starting element expN> [,<elements to sort expN> [, <sort order expN>]]])

<oRef>

A reference to the array you want to sort.

<starting element expN>

In a one-dimensional array, the number of the element in <oRef> at which you want to start sorting. In a two-dimensional array, the number (subscript) of the column on which you want to sort. Without <starting element expN>, *sort*() starts sorting at the first element or column in the array.

<elements to sort expN>

In a one-dimensional array, the number of elements you want to sort. In a two-dimensional array, the number of rows to sort. Without <elements to sort expN>, *sort*() sorts the rows starting at the row containing element <starting element expN> to the last row. If you want to specify a value for <elements to sort expN>, you must also specify a value for <starting element expN>.

<sort order expN>

The sort order:

- 0 specifies ascending order (the default)
- 1 specifies descending order

If you want to specify a value for <sort order expN>, you must also specify values for <elements to sort expN> and <starting element expN>.

Property of

Array

Description

sort() requires that all the elements on which you're sorting be of the same data type. The elements to sort in a one-dimensional array must be of the same data type, and the elements of the column by which rows are to be sorted in a two-dimensional array must be of the same data type.

sort() arranges elements in alphabetical, numerical, chronological, or logical order, depending on the data type of <starting element expN>.

One-dimensional arrays

Suppose you create an array with the following statement:

```
aNums = {5, 7, 3, 9, 4, 1, 2, 8}
```

That creates an array with the elements in this order:

```
5    7    3    9    4    1    2    8
```

If you call aNums.*sort*(0, 5), IntraBuilder sorts the first five elements so that the array elements are in this order:

```
3    4    5    7    9    1    2    8
```

If you then call aNums.*sort*(4, 2), IntraBuilder sorts two elements starting at the fifth element so that the array elements are now in this order:

```
3    4    5    7    1    9    2    8
```

Two-dimensional arrays

Using *sort*() with a two-dimensional array is similar to sorting a table. Array rows correspond to records, and array columns correspond to fields.

When you sort a two-dimensional array, whole rows are sorted, not just the elements in the column where <starting element expN>) is located.

For example, suppose you create the array aInfo and fill it with the following data:

Sep 15 1965	7	A
Dec 31 1965	4	D
Jan 19 1945	8	C
May 2 1972	2	B

If you call aInfo.*sort*(0), IntraBuilder sorts all rows in the array beginning with element number 0. The rows are sorted by the dates in the first column because element 0 is a date. The following figure shows the results.

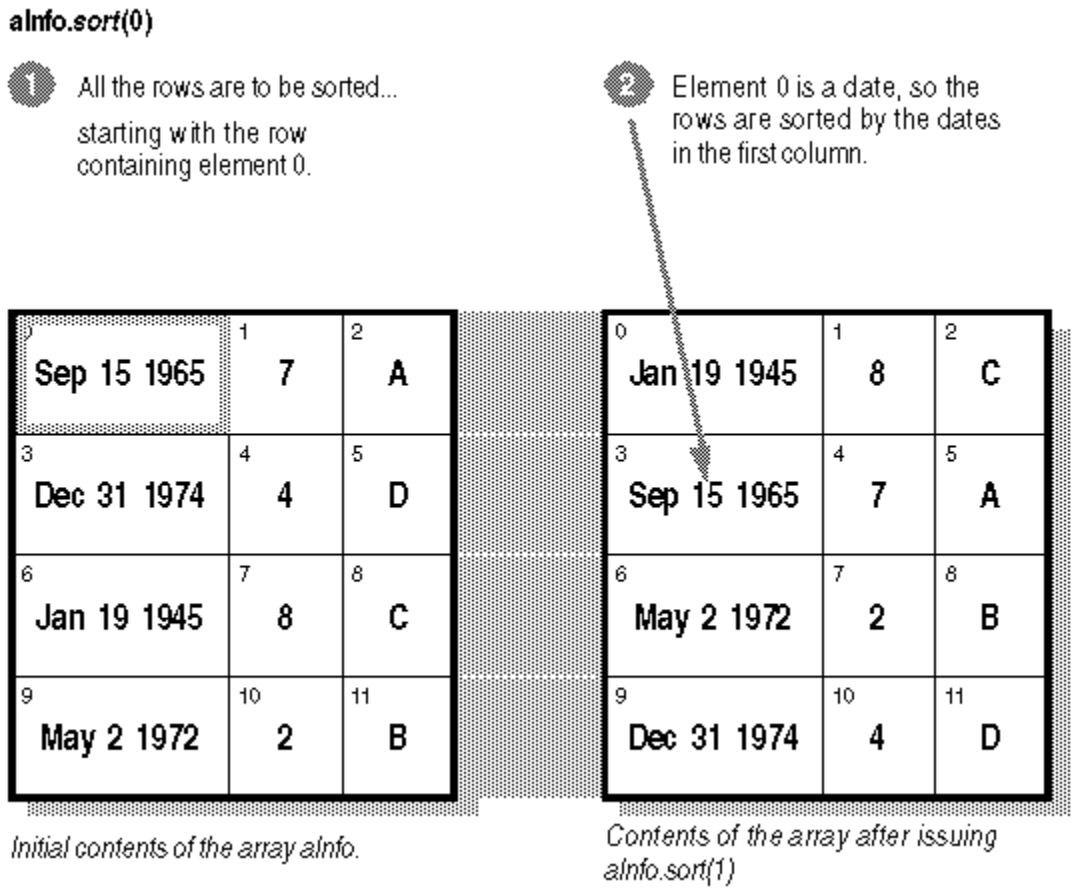


Figure 9.9 **aInfo.***sort* (0)

If you then call aInfo.*sort*(4, 2), IntraBuilder sorts two rows in the array starting with element number 4, whose value is 7. *sort*() sorts the second and the third rows based on the numbers in the second column. The following figure shows the results.

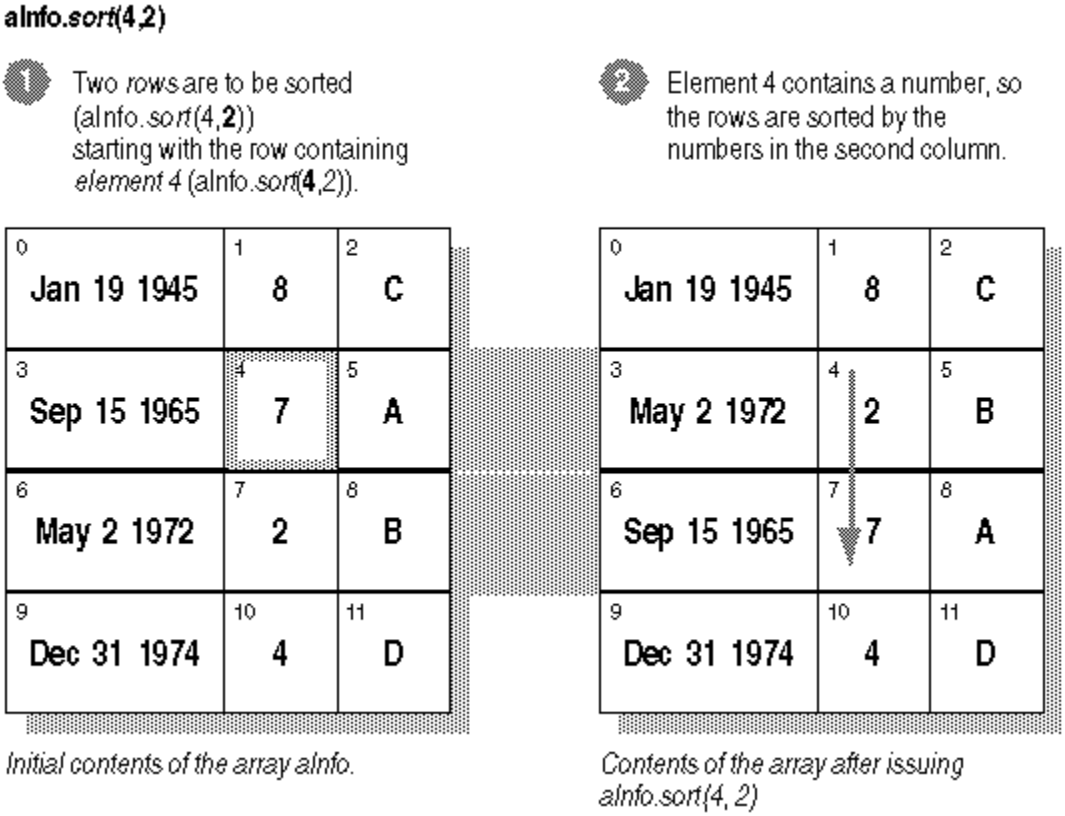


Figure 9.10 **Using** sort() with a two-dimensional array

sort() example

The following example uses *dir*() to store the file information for all the files in the current directory to the array aFiles. Then the array is sorted on the modification date. Manifest constants to represent the columns are created with the #define preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 0  // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 1
#define ARRAY_DIR_DATE 2
#define ARRAY_DIR_TIME 3
#define ARRAY_DIR_ATTR 4
var aFiles = new Array();
var nFiles = aFiles.dir();
aFiles.sort( ARRAY_DIR_SIZE );  // Sort by size
for ( var nFile = 0; nFile < nFiles; nFile++ ) {
  _sys.scriptOut.writeln( aFiles[ nFile, ARRAY_DIR_NAME ] );
  _sys.scriptOut.column = 25;   // Line up column
  _sys.scriptOut.write( aFiles[ nFile, ARRAY_DIR_SIZE ] );
}
```

subscript()

Related topics Example

Returns the row number or the column number of a specified element in an array.

Syntax

<oRef>.*subscript*(<element expN>, <row/column expN>)

<oRef>

A reference to the array.

<element expN>

The element number.

<row/column expN>

A number, either 1 or 2, that determines whether you want to return the row or column subscript of an array. If <row/column expN> is 1, subscript() returns the number of the row subscript. If <row/column expN> is 2, subscript() returns the number of the column subscript.

If <oRef> is a one-dimensional array, IntraBuilder returns an error if <row/column expN> is a value other than 1.

Property of

Array

Description

Use *subscript*() when you know the number of an element in a two-dimensional array and want to reference the element by using its subscripts.

If you need to determine both the row and column number of an element in a two-dimensional array, call *subscript*() twice, once with a value of 1 for <row/column expN> and once with a value of 2 for <row/column expN>. For example, if the element number is in the variable nElement, execute the following statements to get its subscripts:

```
nRow = aExample.subscript( nElement, 1 );
nCol = aExample.subscript( nElement, 2 );
```

In one-dimensional arrays, the number of an element is the same as its subscript, so there is no need to use *subscript*(). For example, if aOne is a one-dimensional array, aOne.*subscript*(3) returns 3, aOne.*subscript*(5) returns 5, and so on.

You can also use *subscript*() to determine the number of rows and columns in a two-dimensional array by getting the subscripts for the last element in the array. For example,

```
nRows = aExample.subscript( aExample.length - 1, 1 ) + 1;
nCols = aExample.subscript( aExample.length - 1, 2 ) + 1;
```

Because array elements and subscripts are zero-based, you must subtract one from the array's *length* to get the last element number and add one to the value returned by *subscript*() to get the number of rows or columns.

subscript() is the inverse of *element*(), which returns an element number when you specify the element subscripts.

subscript() example

The following example displays all the nonzero-size files in your Windows Temp directory. First it tries to find the directory where your temporary files are stored by looking for the DOS environment variable TMP. Then it uses *dir*() to store the file information for all the files in that directory (or the current directory if the TMP directory is not found) to the array aFiles. All the rows that have a file size of zero are deleted using a combination of *scan*(), *subscript*(), and *delete*().

scan() can simply search for zeros because there are no other numeric columns in the array created by *dir*(). If it finds one, *subscript*() is called to return the corresponding row number for the matching element. Then the row number is used in the *delete*() call.

Manifest constants to represent the columns are created with the #define preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 0  // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 1
#define ARRAY_DIR_DATE 2
#define ARRAY_DIR_TIME 3
#define ARRAY_DIR_ATTR 4
// Look for DOS environment variable TMP
var cTempDir = new StringEx( _sys.env.getEnv( "TMP" ) );
// If defined, make sure it has trailing backslash
if ( cTempDir != "" ) {
  if ( cTempDir.right( 1 ) != "\\" ) {  // No trailing backslash
    cTempDir += "\\"                    //   so add one
  }
}
var aFiles = new Array();
var nFiles = aFiles.dir( cTempDir + "\\*.*" );          // Read all files in
TMP dir
while ( ( nElement = aFiles.scan( 0 ) ) >= 0 ) {        // Find zero-byte
files and
  aFiles.delete( aFiles.subscript( nElement, 1 ), 1 );  // delete by row
  nFiles--;                                             // Decrement file
count
}
for ( var nFile = 0; nFile < nFiles; nFile++ ) {        // Display results
  _sys.scriptOut.writeln( aFiles[ nFile, ARRAY_DIR_NAME ] );
  _sys.scriptOut.column = 25;   // Line up column
  _sys.scriptOut.write( aFiles[ nFile, ARRAY_DIR_SIZE ] );
}
```

class File

Example

An object that provides byte-level access to files and contains various file directory methods.

Syntax

[<oRef> =] new File()

<oRef>

A variable or property in which to store a reference to the newly created File object.

Properties

The following tables list the properties and methods of the File class. (No events are associated with this class.)

Property	Default	Description
className	File	Identifies the object as an instance of the File class
handle	–1	Operating system file handle
path		Full path and file name for open file
position	0	Current position of file pointer, relative to the start of the file

Method	Parameters	Description
accessDate()		Returns the last date a file was opened
close()		Closes the currently open file
copy()	<filename expC> , <new name expC>	Makes a copy of the specified file
create()	<filename expC> [,<access rights>]	Creates a new file with optional access attributes
createDate()	<filename expC>	Returns the date when the file was created
createTime()	<filename expC>	Returns the time a file was created as a string
date()	<filename expC>	Returns the date the file was last modified
delete()	<filename expC>	Deletes the specified file
eof()		Returns true or false indicating if the file pointer is positioned past the end of the currently open file
error()		Returns a number indicating the last error encountered
exists()	<filename expC>	Returns true or false to indicate whether the specified disk file exists
flush()		Writes current data in the file buffer to disk and keeps file open
gets()	[<chars read expN>] [, <eol expC>]	Reads and returns a line from a file, leaving the file pointer at the beginning of the next line. Same as readln()
open()	<filename expC> [,<access rights>]	Opens an existing file with optional access attributes
puts()	<input string expC> [, <max chars expN> [, <eol expC>]	Writes a character string and end-of-line character(s) to a file. Same as writeln()
read()	<characters expN>	Reads and returns the specified number of characters from the file starting from the current file pointer position; leaving the file pointer at the character after the last one read
readln()	[<chars read expN>] [, <eol expC>]	Reads and returns a line from a file, leaving the file pointer at the beginning of the next line. Same as gets().
rename()	<filename expC>	Changes the name of the specified file to a new name

	, <new name expC>	
seek()	<offset expN> [, 0 | 1 | 2]	Moves the file pointer the specified number of bytes within a file, optionally allowing the movement to be from the beginning (0), end(2), or current file position (1)
shortName()	<filename expC>	Returns the short (8.3) name for a file
size()	<filename expC>	Returns the number of bytes in the specified file
time()	<filename expC>	Returns the time the file was last modified as a string
write()	<expC> [, <max chars expN>]	Writes the specified string into the file at the current file position, overwriting any existing data and leaving the file pointer at the character after the last character written
writeln()	<input string expC> [, <max chars expN> [, <eol expC>]	Writes a character string and end-of-line character(s) to a file. Same as puts().

d Description

Use a File object for direct byte-level access to files. Once you create a new File object, you can *open*() an existing file or *create*() a new one. Be sure to *close*() the file when you are done. A File object may access only one file at a time, but after closing a file, you may open or create another.

File objects also contain information and utility methods for file directories, such as returning the size of a file or changing a file name. If you intend to call multiple methods, you can create and reuse a File object. For example,

```
var ff = new File();
_sys.scriptOut.writeln( ff.size( "INTRA.HLP" ));
_sys.scriptOut.writeln( ff.accessDate( "INTRA.HLP" ));
```

Or you can create a File object for a *with* block. For example,

```
with ( new File() ) {
  _sys.scriptOut.writeln( ff.size( "INTRA.HLP" ));
  _sys.scriptOut.writeln( ff.accessDate( "INTRA.HLP" ));
}
```

For a single call, you can create and use the File object in the same statement:

```
_sys.scriptOut.writeln( new File().size( "INTRA.HLP" ));
```

class File example

Suppose you have a data file generated by a mainframe computer that has fixed length records with no record breaks. You want to convert this file so that you have one record on each line. Use two File objects to read and write the file, adding line breaks as you write:

```
#define REC_LENGTH 80
#define IN_FILE    "STUFF.REC"
#define OUT_FILE   "STUFF.TXT"
fIn  = new File();
fOut = new File();
fIn.open( IN_FILE );
fOut.create( OUT_FILE );
while ( !fIn.eof() ) {
  fOut.writeln( fIn.read( REC_LENGTH ));  // Read fixed length; write with
line break
}
fIn.close();
fOut.close();
```

accessDate()

Related topics Example

Returns the last date a file was opened.

Syntax

<oRef>.*accessDate*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

accessDate() checks the file specified by <filename expC> and returns the date that the file was last opened by the operating system for reading or writing.

To get the date the file was last modified, use *date*(). For the date the file was created, use *createDate*().

accessDate() example

The following example uses *accessDate*() to display the last date the Help file was used:

```
? new File().accessDate( "C:/Program
Files/Borland/Intrabuilder/Bin/INTRA.HLP" );
```

close()

Related topics Example

Closes a file previously opened with *create*() or *open*().

Syntax

<oRef>.*close*()

<oRef>

A reference to the File object that created or opened the file.

Property of

File

Description

close() closes a file you've opened with *create*() or *open*(). *close*() returns *true* if it's able to close the file. If the file is no longer available (for example, the file was on a floppy disk that has been removed) and there is data in the buffer that has not yet been written to disk, *close*() returns *false*.

Always close the file when you're done with it.

To save the file to disk without closing it, use *flush*().

close() example

The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"
var f = new File();
if ( f.exists( LOG_FILE ) ) {
  f.open( LOG_FILE, "A" );
}
else {
  f.create( LOG_FILE, "A" );
}
f.writeln( new Date().toLocaleString() );
f.close();
```

copy()

Related topics Example

Duplicates a specified file.

Syntax

<oRef>.*copy*(<filename expC>, <new name expC>)

<oRef>

A reference to a File object.

<filename expC>

Identifies the file to duplicate (also known as the source file). <filename expC> may be a file name skeleton with wildcard characters. In that case, IntraBuilder displays a dialog box in which you select the file to duplicate.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

<new name expC>

Identifies the target file that will be created or overwritten by *copy*(). <new name expC> may be a filename skeleton with wildcard characters. In that case, IntraBuilder displays a dialog box in which you specify the name of the target file and its directory.

Property of

File

Description

copy() lets you duplicate an existing file at the operating system level. *copy*() duplicates a single file of any type.

Any existing file with the same name is overwritten without warning.

copy() does not automatically copy the auxiliary files associated with table files, such as indexes and memo files. For example, it does not copy the MDX or DBT file associated with a DBF file. When copying tables, use the Database object's *copyTable*() method.

You cannot *copy*() a file that has been opened for writing with the *open*() or *create*() methods; it must be closed first.

copy() example

The following example makes a copy of a file in the current directory:

```
new File().copy( "AFILE", "ACOPY" );
```

create()

Related topics Example

Creates and opens a specified file.

Syntax

<oRef>.*create*(<filename expC>[, <access expC>])

<oRef>

A reference to a File object.

<filename expC>

The name of the file to create and open. By default, *create*() creates the file in the current directory. To create the file in another directory, specify a full path name for <filename expC>.

<access expC>

The access level of the file to create, as shown in the following table. The access level string is not case-sensitive. If omitted, the default is read and write. Append is a more restrictive version of write; the data is always added to the end of the file.

<access expC>	**Access level**
"R"	Read-only
"W"	Write-only
"A"	Append-only
"RW" or "WR"	Read and write
"RA" or "AR"	Read and append

Property of

File

Description

Use *create*() to create a file with a name you specify, assign the file the level of access you specify, and open the file. If IntraBuilder can't create the file (for example, if the file is already open), an exception occurs.

If <filename expC> already exists, it is overwritten without warning. To see if a file with the same name already exists, use *exists*() before issuing *create*().

To use other File methods, such as *read*() and *write*(), first open a file with *create*() or *open*().

When you open a file with *create*(), the file is empty, so the file pointer is positioned at the first character in the file. Use *seek*() to position the file pointer before reading from or writing to a file.

create() example

The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"
var f = new File();
if ( f.exists( LOG_FILE ) ) {
  f.open( LOG_FILE, "A" );
}
else {
  f.create( LOG_FILE, "A" );
}
f.writeln( new Date().toLocaleString() );
f.close();
```

createDate()

Related topics Example

Returns the date a file was created.

Syntax

<oRef>.*createDate*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

createDate() checks the file specified by <filename expC> and returns the date that the file was created.

To get the date the file was last modified, use *date*(). For the date the file was last accessed, use *accessDate*(). To get the time the file was created, use *createTime*().

createDate() example

The following example uses *createDate*() to display the date the Help file was created:

```
? new File().createDate( "C:/Program
Files/Borland/Intrabuilder/Bin/INTRA.HLP" );
```

createTime()

Related topics Example

Returns the time a file was created.

Syntax

<oRef>.*createTime*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

createTime() checks the file specified by <filename expC> and returns the time, as a character string, that the file was created.

To get the date the file was created, use *createDate*().

createTime() example

The following example uses *createTime*() to display the time the Help file was created:

```
? new File().createTime( "C:/Program
Files/Borland/Intrabuilder/Bin/INTRA.HLP" );
```

date()

Related topics Example

Returns the date stamp for a file, the date the file was last modified.

Syntax

<oRef>.*date*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

Use *date*() to determine the date on which the last change was made to a file on disk.

When you update a file, IntraBuilder changes the file's date stamp to the current operating system date when the file is written to disk. For example, when the user edits a DB table, IntraBuilder changes the date stamp on the table file when the file is closed. *date*() reads the date stamp and returns its current value.

To get the date the file was created, use *createDate*(). For the date the file was last accessed, use *accessDate*(). To get the time the file was last changed, use *time*().

createTime() example

The following example uses *date*() to display the date IntraBuilder's INI file was last modified:

```
? new File().date( "C:/Program Files/Borland/Intrabuilder/Bin/INTRA.INI" );
```

delete()

Related topics Example

Removes a file from a disk.

Syntax

<oRef>.*delete*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

Identifies the file to remove. <filename expC> may be a filename skeleton with wildcard characters. In that case, IntraBuilder displays a dialog box in which you select the file to duplicate.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

delete() deletes a file from a disk.

delete() does not automatically remove the auxiliary files associated with table files, such as indexes and memo files. For example, it does not delete the MDX or DBT files associated with a DBF file. When deleting tables, use the Database object's *dropTable*() method.

delete() example

The following examples deletes a file in the current directory:

```
new File().delete( "AFILE" );
```

eof()

Related topics Example

Returns *true* if the file pointer is at the end of a file previously opened with *create*() or *open*()

Syntax

<oRef>.*eof*()

<oRef>

A reference to the File object that created or opened the file.

Property of

File

Description

eof() determines if the file pointer of the file you specify is at the end of the file (EOF), and returns *true* if it is and *false* if it is not. The file pointer is considered to be at EOF if it is positioned at the byte after the last character in the file.

You can move the file pointer to the end of the file with *seek*(). If a file is empty, as it is when you first create a new file with *create*(), *eof*() returns *true*.

eof() example

Suppose you have a data file generated by a mainframe computer that has fixed-length records with no record breaks. You want to convert this file so that you have one record on each line. Use two File objects to read and write the file, adding line breaks as you write:

```
#define REC_LENGTH 80
#define IN_FILE    "STUFF.REC"
#define OUT_FILE   "STUFF.TXT"
fIn  = new File();
fOut = new File();
fIn.open( IN_FILE );
fOut.create( OUT_FILE );
while ( !fIn.eof() ) {
  fOut.writeln( fIn.read( REC_LENGTH ));  // Read fixed length; write with
line break
}
fIn.close();
fOut.close();
```

error()

Related topics

Returns the error number of the most recent byte-level input or output error, or 0 if the most recent byte-level method was successful.

Syntax

<oRef>.*error*()

<oRef>

A reference to the File object that attempted the operation.

Property of

File

Description

To trap errors, call the File object method in a *try* block. Use the number that *error*() returns in a *catch* block to respond to errors in the byte-level methods of the File object. The following table lists the byte-level method errors that *error*() returns.

Error number_	Cause of error
2	File or directory not found
3	Bad path name
4	No more file handles available
5	Can't access file
6	Bad file handle
8	No more directory entries available
9	Error trying to set the file pointer
13	No more disk space
14	End of file

exists()

Related topics Example

Tests for the existence of a file. Returns *true* if the file exists and *false* if it doesn't.

Syntax

<oRef>.*exists*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to search for. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension.

Property of

File

Description

Use *exists*() to determine whether a file exists. You can use either the long file name or the short file name.

exists() example

The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"
var f = new File();
if ( f.exists( LOG_FILE ) ) {
  f.open( LOG_FILE, "A" );
}
else {
  f.create( LOG_FILE, "A" );
}
f.writeln( new Date().toLocaleString() );
f.close();
```

flush()

Related topics

Writes to disk a file previously opened with *create*() or *open*() without closing the file. Returns *true* if successful and *false* if unsuccessful.

Syntax

<oRef>.*flush*()

<oRef>

A reference to the File object that created or opened the file.

Property of

File

Description

Use *flush*() to save a file in the file buffer to disk, flush the file buffer, and keep the file open. If *flush*() is successful, it returns *true*.

Flushing a buffer to disk is similar to saving the file and continuing to work on it. Until you flush an open file buffer to disk, any data in the buffer is stored only in RAM (random-access memory). If the power to the computer fails or IntraBuilder ends abnormally, data in RAM is lost. However, if you have used *flush*() to write the file buffer to disk, you lose only data that was added between the time you issued *flush*() and the time the system failed.

To save the file to disk and close the file, use *close*().

gets()

Related topics

Returns a line of text from a file previously opened with *create*() or *open*().

Syntax

<oRef>.*gets*([<characters expN> [, <end-of-line expC>]])

Property of

File

Description

gets() is identical to *readln*(). See *readln*() for details.

handle

Related topics

The operating system file handle for a file previously opened with *create*() or *open*().

Property of

File

Description

When a file is opened by the operating system, it is assigned a file handle, an arbitrary number that identifies that open file. Applications then use that file handle to refer to that file.

A File object's *handle* property reflects the file handle used by IntraBuilder to access a file opened with *create*() or *open*(). It is a read-only property and is generally informational only. By calling methods of the File object, IntraBuilder internally uses the file handle to perform its operations.

open()

Related topics Example

Opens a specified file.

Syntax

<oRef>.*open*(<filename expC>[, <access expC>])

<oRef>

A reference to a File object.

<filename expC>

The name of the file to open. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

<access expC>

The access level of the file being opened, as shown in the following table. The access level string is not case-sensitive. If omitted, the default is read-only. Append is a more restrictive version of write; the data is always added to the end of the file

<access expC>	**Access level**
“R”	Read-only
“W”	Write-only
“A”	Append-only
“RW” or “WR”	Read and write
“RA” or “AR”	Read and append

Property of

File

Description

Use *open*() to open a file with a name you specify and assign the file the level of access you specify. If IntraBuilder can’t open the file (for example, if the file is already open), an exception occurs.

To use other File methods, such as *read*() and *write*(), first open a file with *open*() or *create*().

If you open the file with append-only or read and append access, the file pointer is positioned at the end-of-file, after the last character. For other access levels, the file pointer is positioned at the first character in the file. Use *seek*() to position the file pointer before reading from or writing to a file.

open() example

The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"
var f = new File();
if ( f.exists( LOG_FILE ) ) {
  f.open( LOG_FILE, "A" );
}
else {
  f.create( LOG_FILE, "A" );
}
f.writeln( new Date().toLocaleString() );
f.close();
```

path

Related topics

The full path and file name for a file previously opened with *create*() or *open*().

Property of

File

Description

When you open a file with *create*() or *open*(), the path is optional. If you use *create*() without a path, the file is created in the current directory. If you use *open*() without a path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any.

A File object's *path* property reflects the full path and file name for the open file. It is a read-only property.

position

Related topics

The position of the file pointer in a file previously opened with *create*() or *open*().

Property of

File

Description

A File object's *position* property reflects the current position of the file pointer.
It is a read-only property. To move the file pointer, use *seek*(). Reading and writing to a file also moves the file pointer.

The position is zero-based. The first character in the file is at position zero.

puts()

Related topics

Writes a character string and one or two end-of-line characters to a file previously opened with *create*() or *open*(). Returns the number of characters written.

Syntax

<oRef>.*puts*(<string expC> [, <characters expN> [, <end-of-line expC>]])

Property of

File

Description

puts() is identical to *writeln*(). See *writeln*() for details.

read()

Related topics Example

Returns a specified number of characters from a file previously opened with *create*() or *open*().

Syntax

<oRef>.*read*(<characters expN>)

<oRef>

A reference to the File object that created or opened the file.

<characters expN>

The number of characters to return from the specified file.

Property of

File

Description

read() returns the number of characters you specify from the file opened by the File object. *read*() starts reading characters from the current file pointer position, leaving the file pointer at the character immediately after the last character read. Use *seek*() to move the file pointer before or after you use *read*().

If the file to be read is a text file, use *readln*() instead. *readln*() looks for end-of-line characters, and returns the contents of the line, without the end-of-line character(s).

To write to a file, use *write*().

read() example

Suppose you have a data file generated by a mainframe computer that has fixed-length records with no record breaks. You want to convert this file so that you have one record on each line. Use two File objects to read and write the file, adding line breaks as you write:

```
#define REC_LENGTH 80
#define IN_FILE    "STUFF.REC"
#define OUT_FILE   "STUFF.TXT"
fIn  = new File();
fOut = new File();
fIn.open( IN_FILE );
fOut.create( OUT_FILE );
while ( !fIn.eof() ) {
  fOut.writeln( fIn.read( REC_LENGTH ));  // Read fixed length; write with
line break
}
fIn.close();
fOut.close();
```

readln()

Related topics Example

Returns a line of text from a file previously opened with *create*() or *open*().

Syntax

<oRef>.*readln*([<characters expN> [, <end-of-line expC>]])

<oRef>

A reference to the File object that created or opened the file.

<characters expN>

The number of characters to read and return before a carriage return is reached.

<end-of-line expC>

The end-of-line indicator, which can be a string of one or two characters. If omitted, the default is a hard carriage return and line feed. The following table lists standard codes used as end-of-line indicators.

Character code (decimal)	(hexadecimal)	Represents
chr(141)	0x8D	Soft carriage return (U.S.)
chr(255)	0xFF	Soft carriage return (Europe)
chr(138)	0x8A	Soft linefeed (U.S.)
chr(0)	0x00	Soft linefeed (Europe)
chr(13)	0x0D	Hard carriage return
chr(10)	0x0A	Hard linefeed

Use the StringEx object's *chr*() method to create the <end-of-line expC> if needed. To designate the <end-of-line expC>, you must also specify the <characters expN>. If you don't want a line length limit, use an arbitrarily high number. For example,

```
cLine = f.readln( 10000, new StringEx().chr( 0x8d ) ); // Soft carriage
return (U.S.)
```

Property of

File

Description

Use *readln*() to read lines from a text file. *readln*() reads and returns a character string from the file opened by the File object, starting at the file pointer position, and reading past but not returning the first end-of-line character(s) it encounters.

readln() will read characters until it encounters the end-of-line character(s) or it reads the number of characters you specify with <characters expN>, whichever comes first. If a file does not have end-of-line character(s) and you do not specify <characters expN>, *readln*() will read and return everything from the current file pointer position to the end of the file.

If the file pointer position is at an end-of-line character(s), *readln*() returns an empty string (“”); the line is empty.

If *readln*() encounters an end-of-line character(s), it positions the file pointer at the character after the end-of-line character(s); that is, at the beginning of the next line. Otherwise, *readln*() positions the file pointer at the character after the last character it returns. Use *seek*() to move the file pointer before or after using *readln*().

If the file being read is not a text file, use *read*() instead. *read*() requires <characters expN> to be specified, and does not treat end-of-line characters specially.

To write a text file, use *writeln*(). *gets*() is identical to *readln*().

readln() example

The following statements display the contents of a text file in an HTML component, replacing the line breaks in the text file with the HTML
 tag. The name of the file is typed into a Text component named text1, and the HTML component is named html1.

```
var f = new File();                                  // Create File object
if ( f.exists( this.form.text1.value ) ) {           // Make sure file exists
  f.open( this.form.text1.value );
  this.form.html1.text = "";                         // Clear HTML component
  while ( !f.eof() ) {
    this.form.html1.text += f.readln() + "<BR>";  // Write lines to HTML
component
  }
  f.close();                                         // Close file
}
else {
  this.form.html1.text = this.form.text1.value + " not found";
}
```

rename()

Related topics Example

Renames a file on disk.

Syntax

<oRef>.*rename*(<filename expC>, <new name expC>)

<oRef>

A reference to a File object.

<filename expC>

Identifies the file to rename (also known as the source file). <filename expC> may be a file name skeleton with wildcard characters. In that case, IntraBuilder displays a dialog box in which you select the file to rename.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

<new name expC>

Identifies the new name for the source file (also known as the target file). <new name expC> may be a file name skeleton with wildcard characters. In that case, IntraBuilder displays a dialog box in which you specify the name of the target file and its directory.

Property of

File

Description

rename() lets you change the name of a file at the operating system level.

If a file exists with the same name as the target file, an exception occurs, and the target file is not overwritten.

If you specify a different drive or directory for the target file, IntraBuilder moves the source file to that location.

rename() does not automatically rename the auxiliary files associated with table files, such as indexes and memo files. For example, it does not rename the MDX or DBT files associated with a DBF file. When renaming tables, use the Database object's *renameTable*() method.

rename() example

The following example changes the name of a file in the current directory to something else:

```
new File().rename( "AFILE", "SOMETHING" );
```

seek()

Related topics Example

Moves the file pointer in a file previously opened with *create*() or *open*(), and returns the new position of the file pointer.

Syntax

<oRef>.*seek*(<offset expN> [, <position expN>])

<oRef>

A reference to the File object that created or opened the file.

<offset expN>

The number of bytes to move the file pointer in the specified file. If <offset expN> is negative, the file pointer moves toward the beginning of the file. If <offset expN> is 0, the file pointer moves to the position you specify with <position expN>. If <offset expN> is positive, the file pointer moves toward the end of the file or beyond the end of the file.

<position expN>

The number 0, 1, or 2, indicating a position relative to the beginning of the file (0), to the file pointer's current position (1), or to the end of the file (2). The default is 0.

Property of

File

Description

seek() moves the file pointer in the file you specify relative to the position specified by <position_expN>, and returns the resulting position of the file pointer as an offset from the beginning of the file. The File object's *position* property is also updated with this new position.

The movement of the file pointer is relative to the beginning of the file unless you specify otherwise with <position expN>. For example, *seek*(5) moves the file pointer five characters from the beginning of the file (the 6th character) while *seek*(5,1) moves it five characters forward from its current position. You can move the file pointer beyond the end of the file, but you can't move it before the beginning of the file.

To move the file pointer to the beginning of a file, use *seek*(0). To move it to the end of a file, use *seek*(0, 2). To move to the last character in a file, use *seek*(–1,2).

read(), *readln*(), *write*(), and *writeln*() also move the file pointer as they read from or write to the file.

seek() example

Suppose you're exporting data from a table in a special format for another program. The export file must have the number of rows of data written in the file, starting at the 9th character. You extend the File class, adding methods to create the export file, write the data in the special format, and record the number of rows written. The following is the method that records the number of rows.

```
function recordRowsWritten()
{
  this.seek( 8 );                          // 9th character == offset 8
  this.write( "" + this.rowsExported );  // Convert number to string to write
}
```

shortName()

Related topics

Returns the short (8.3) name of a file.

Syntax

<oRef>.*shortName*(<filename expC>)

<oRef>

A reference to a File object.

<*filename expC>*

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

shortName() checks the file specified by <filename expC> and returns a name for the file following the DOS file-naming convention (eight-character file name, three-character extension).

size()

Related topics Example

Returns the size of a file in bytes.

Syntax

<oRef>.*size*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

Use *size*() to determine the size of a file on disk.

With the byte-level access methods of the File object, IntraBuilder doesn't update the size on the file recorded on the disk until you *close*() the file.

size() example

The following example uses *size*() to display the size of the Help file:

```
? new File().size( "C:/Program Files/Borland/Intrabuilder/Bin/INTRA.HLP" );
```

time()

Related topics Example

Returns the time stamp for a file, the time the file was last modified.

Syntax

<oRef>.*time*(<filename expC>)

<oRef>

A reference to a File object.

<filename expC>

The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, IntraBuilder looks for the file in the current directory, then in the supplemental search path(s) you specified in the IntraBuilder Explorer, if any. If you specify a file without including its extension, IntraBuilder assumes no extension. If the named file cannot be found, an exception occurs.

Property of

File

Description

Use *time*() to determine the time of day when the last change was made to a file on disk. *time*() returns the time as a character string.

When you update a file, IntraBuilder changes the file's time stamp to the current operating system time when the file is written to disk. For example, when the user edits a DB table, IntraBuilder changes the time stamp on the table file when the file is closed. *time*() reads the time stamp and returns its current value.

To get the time the file was created, use *createTime*(). For the date the file was last modified, use *date*().

time() example

The following example uses *time*() to display the time IntraBuilder's INI file was last modified:

```
? new File().time( "C:/Program Files/Borland/Intrabuilder/Bin/INTRA.INI" );
```

write()

Related topics Example

Writes a character string to a file previously opened with *create*() or *open*(). Returns the number of characters written.

Syntax

<oRef>.*write*(<expC> [, <characters expN>])

<oRef>

A reference to the File object that created or opened the file.

<expC>

The character expression to write to the specified file. If you want to write only a portion of <string expC> to the file, use the <characters expN> argument.

<characters expN>

The number of characters of the specified character expression <string expC> to write to the specified file, starting at the first character in <string expC>. If omitted, the entire string is written.

Property of

File

Description

write() writes a character string to a file. If the file was opened in append-only or read and append mode, the string is always added to the end of the file. Otherwise, the string is written starting at the current file pointer position, overwriting any existing characters. You must have either write or append access to use *write*().

write() returns the number of bytes written to the file. If *write*() returns 0, no characters were written. Either <expC> is an empty string, or the write was unsuccessful.

Use *error*() to determine if an error occurred.

When *write*() finishes executing, the file pointer is located at the character immediately after the last character written. Use *seek*() to move the file pointer before or after you use *write*().

To write to a text file, use *writeln*(). *writeln*() automatically adds the end-of-line character(s).

To read from a file, use *read*().

write() example

Suppose you're exporting data from a table in a special format for another program. The export file must have the number of rows of data written in the file, starting at the 9th character. You extend the File class, adding methods to create the export file, write the data in the special format, and record the number of rows written. The following is the method that records the number of rows.

```
function recordRowsWritten()
{
  this.seek( 8 );                          // 9th character == offset 8
  this.write( "" + this.rowsExported );  // Convert number to string to write
}
```

writeln()

Related topics Example

Writes a character string and one or two end-of-line characters to a file previously opened with *create*() or *open*(). Returns the number of characters written.

Syntax

<oRef>.*writeln*(<string expC> [, <characters expN> [, <end-of-line expC>]])

<oRef>

A reference to the File object that created or opened the file.

<string expC>

The character expression to write to the specified file. If you want to write only a portion of <string expC> to the file, use the <characters expN> argument.

<characters expN>

The number of characters of the specified character expression <string expC> to write to the specified file, starting at the first character in <string expC>. If omitted, the entire string is written.

<end-of-line expC>

The end-of-line indicator, which can be a string of one or two characters. If omitted, the default is a hard carriage return and line feed. The following table lists standard codes used as end-of-line indicators.

Character code (decimal)	(hexadecimal)	Represents
chr(141)	0x8D	Soft carriage return (U.S.)
chr(255)	0xFF	Soft carriage return (Europe)
chr(138)	0x8A	Soft linefeed (U.S.)
chr(0)	0x00	Soft linefeed (Europe)
chr(13)	0x0D	Hard carriage return
chr(10)	0x0A	Hard linefeed

Use the StringEx object's *chr*() method to create the <end-of-line expC> if needed. To designate the <end-of-line expC>, you must also specify the <characters expN>. If you don't want a line length limit, use an arbitrarily high number. For example,

```
f.writeln( cLine, 10000, new StringEx().chr( 0x8d ) ); // Soft carriage
return (U.S.)
```

Property of

File

Description

Use *writeln*() to write text files. *writeln*() writes a character string and one or two end-of-line characters to a file. If the file was opened in append-only or read and append mode, the string is always added to the end of the file. Otherwise, the string is written starting at the current file pointer position, overwriting any existing characters. You must have either write or append access to use *writeln*().

writeln() returns the number of bytes written to the file, including the end-of-line character(s). If *writeln*() returns 0, no characters were written. Either <string expC> is an empty string, or the write was unsuccessful.

Use *error*() to determine if an error occurred.

When *writeln*() finishes executing, the file pointer is located at the character immediately after the last character written, which is the end-of-line character. Successive *writeln*() calls writes one line after another. Use *seek*() to move the file pointer before or after you use *writeln*().

To write to a file that is not a text file, use *write*(). *write*() does not add the end-of-line character(s).

To read from a text file, use *readln*(). *puts*() is identical to *writeln*().

writeln() example

The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"
var f = new File();
if ( f.exists( LOG_FILE ) ) {
  f.open( LOG_FILE, "A" );
}
else {
  f.create( LOG_FILE, "A" );
}
f.writeln( new Date().toLocaleString() );
f.close();
```

Local SQL overview

Related topics

The Borland Database Engine (BDE) enables access to database tables through the industry-standard SQL language. Different table formats, for example InterBase¨ and Oracle, use different dialects of SQL. Local SQL (sometimes called “client-based SQL”) is a subset of ANSI-92 SQL for accessing DB (Paradox) and DBF (dBASE) tables and fields (called “columns” in SQL).

Although it is called “local” SQL, the DB and DBF tables may reside on a remote network file server.

For information on the SQL dialect for other table formats, consult your SQL server documentation.

SQL statements are divided into two categories:

- Data definition language

These statements are used for creating, altering, and dropping tables, and for creating and dropping indexes.

- Data manipulation language

These statements are used for selecting, inserting, updating, and deleting table data.

In the examples, an SQL statement may be displayed on multiple lines for readability. But SQL is not line-oriented. When an SQL statement is specified in a string, as it is in a Query object’s *sql* property, the entire SQL statement is specified in a single line.

Although JavaScript is a case-sensitive language, SQL is not. The convention for SQL keywords is all uppercase, which is used in this series of Help topics. SQL statements in the rest of the *Language Reference* may use either uppercase or lowercase.

Naming conventions

Related topics

This section describes the naming conventions for tables and columns in local SQL.

Tables

Local SQL supports full file and path specifications for table names. Table names with a path, spaces, or other special characters in their names must be enclosed in single or double quotation marks. If the SQL statement is typed as a literal string, all backslashes must be doubled, because the backslash acts as the escape character in JavaScript strings; or you can use forward slashes instead of backslashes. For example,

```
SELECT * FROM PARTS.DB               // Simple name with extension; no quotes
required
SELECT * FROM "AIRCRAFT PARTS.DB"    // Name has space; quotes needed
SELECT * FROM "C:\\SAMPLE\\PARTS.DB" // Backslash doubled
SELECT * FROM "C:/SAMPLE/PARTS.DB"   // Forward slash instead of backslash
```

Local SQL also supports BDE aliases for table names. For example,

```
SELECT * FROM :IBAPPS:KBCAT
```

If you omit the file extension for a local table name, the table is assumed to be the table type specified the BDE Configuration Utility, either in the Default Driver setting on the System page or in the default drive type for the standard alias associated with the query or table.

Finally, local SQL permits table names to duplicate SQL keywords as long as those table names are enclosed in single or double quotation marks. For example,

```
SELECT PASSID FROM "PASSWORD"
```

Columns

Local SQL supports multi-word column names and column names that duplicate SQL keywords as long as those column names are

- Enclosed in single or double quotation marks
- Prefaced with an SQL table name or table correlation name

For example, the following column name is two words:

```
SELECT E."Emp Id" FROM EMPLOYEE E
```

In the next example, the column name duplicates the SQL DATE keyword:

```
SELECT DATELOG."DATE" FROM DATELOG
```

Operators

Related topics

Local SQL supports the following operators:

Local SQL operators

Type	Operator	Type	Operator
Arithmetic	+ – * /	Logical	AND OR NOT
Comparison	< > = <> >= <= IS NULL IS NOT NULL	String concatenation	||

Reserved words

Related topics

The following is an alphabetical list of the 215 words reserved by local SQL:

List of local SQL reserved words

ACTIVE	ADD	ALL	AFTER
ALTER	AND	ANY	AS
ASC	ASCENDING	AT	AUTO
AUTOINC	AVG	BASE_NAME	BEFORE
BEGIN	BETWEEN	BLOB	BOOLEAN
BOTH	BY	BYTES	CACHE
CAST	CHAR	CHARACTER	CHECK
CHECK_POINT_LENGTH	COLLATE	COLUMN	COMMIT
COMMITTED	COMPUTED	CONDITIONAL	CONSTRAINT
CONTAINING	COUNT	CREATE	CSTRING
CURRENT	CURSOR	DATABASE	DATE
DAY	DEBUG	DEC	DECIMAL
DECLARE	DEFAULT	DELETE	DESC
DESCENDING	DISTINCT	DO	DOMAIN
DOUBLE	DROP	ELSE	END
ENTRY_POINT	ESCAPE	EXCEPTION	EXECUTE
EXISTS	EXIT	EXTERNAL	EXTRACT
FILE	FILTER	FLOAT	FOR
FOREIGN	FROM	FULL	FUNCTION
GDSCODE	GENERATOR	GEN_ID	GRANT
GROUP	GROUP_COMMIT_WAIT_TIME	HAVING	HOUR
IF	IN	INT	INACTIVE
INDEX	INNER	INPUT_TYPE	INSERT
INTEGER	INTO	IS	ISOLATION
JOIN	KEY	LONG	LENGTH
LOGFILE	LOWER	LEADING	LEFT
LEVEL	LIKE	LOG_BUFFER_SIZE	MANUAL
MAX	MAXIMUM_SEGMENT	MERGE	MESSAGE
MIN	MINUTE	MODULE_NAME	MONEY
MONTH	NAMES	NATIONAL	NATURAL
NCHAR	NO	NOT	NULL
NUM_LOG_BUFFERS	NUMERIC	OF	ON
ONLY	OPTION	OR	ORDER
OUTER	OUTPUT_TYPE	OVERFLOW	PAGE_SIZE
PAGE	PAGES	PARAMETER	PASSWORD
PLAN	POSITION	POST_EVENT	PRECISION
PROCEDURE	PROTECTED	PRIMARY	PRIVILEGES
RAW_PARTITIONS	RDB$DB_KEY	READ	REAL
RECORD_VERSION	REFERENCES	RESERV	RESERVING
RETAIN	RETURNING_VALUES	RETURNS	REVOKE
RIGHT	ROLLBACK	SECOND	SEGMENT
SELECT	SET	SHARED	SHADOW

SCHEMA	SINGULAR	SIZE	SMALLINT
SNAPSHOT	SOME	SORT	SQLCODE
STABILITY	STARTING	STARTS	STATISTICS
SUB_TYPE	SUBSTRING	SUM	SUSPEND
TABLE	THEN	TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE	TO	TRAILING
TRANSACTION	TRIGGER	TRIM	UNCOMMITTED
UNION	UNIQUE	UPDATE	UPPER
USER	VALUE	VALUES	VARCHAR
VARIABLE	VARYING	VIEW	WAIT
WHEN	WHERE	WHILE	WITH
WORK	WRITE	YEAR	

Data definition

Related topics

Local SQL supports data definition language (DDL) for creating, altering, and dropping tables, and for creating and dropping indexes.

Local SQL does not permit the substitution of parameters for values in DDL statements.

The following DDL statements are supported:

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- CREATE INDEX
- DROP INDEX

Data manipulation

Related topics

This section describes functions available to data manipulation language (DML) statements in local SQL. It covers

- Parameter substitutions in DML statements
- Aggregate functions
- String functions
- Date function
- Updatable queries

With some restrictions, local SQL supports the following statements for data manipulation:

- SELECT, for retrieving existing data
- INSERT, for adding new data to a table
- UPDATE, for modifying existing data
- DELETE, for removing existing data from a table

Parameter substitutions in DML statements

Related topics

Parameters can be used in DML statements in place of values. Parameters must always be preceded by a colon (:). For example,

```
SELECT LAST_NAME, FIRST_NAME
   FROM "CUSTOMER.DB"
   WHERE LAST_NAME > :parm1 AND FIRST_NAME < :parm2
```

Assigning an SQL statement with parameters in a Query or StoredProc object automatically creates the corresponding elements in the object's params array. You then store values to substitute in that array.

Aggregate functions

Related topics

The following ANSI-standard SQL aggregate functions are available to local SQL for use with data retrieval:

- SUM(), for totaling all numeric values in a column
- AVG(), for averaging all non-NULL numeric values in a column
- MIN(), for determining the minimum value in a column
- MAX(), for determining the maximum value in a column
- COUNT(), for counting the number of values in a column that match specified criteria

Complex aggregate expressions are supported, such as

```
SUM( Field * 10 )
SUM( Field ) * 10
SUM( Field1 + Field2 )
```

String functions

Related topics

Local SQL supports the following ANSI-standard SQL string manipulation functions for retrieval, insertion, and updating:

- UPPER(), to force a string to uppercase
- LOWER(), to force a string to lowercase
- TRIM(), to remove repetitions of a specified character from the left, right, or both sides of a string
- SUBSTRING() to create a substring from a string

Substring

Related topics

SUBSTRING() takes a string and creates a substring of that string. The following query returns the first 10 characters of the CUSTNAME column.

```
SELECT SUBSTRING( CUSTNAME FROM 1 FOR 10 ) FROM CUSTOMER
```

Date function

Related topics

Local SQL supports the EXTRACT() function for isolating a single numeric field from a date/time field on retrieval using the following syntax:

```
EXTRACT (<extract field> FROM <field name>)
```

Date function example

The following statement extracts the year value from a DATE field:

```
SELECT EXTRACT(YEAR FROM HIRE_DATE)
  FROM EMPLOYEE
```

You can also extract MONTH, DAY, HOUR, MINUTE, and SECOND using this function.

In local SQL, EXTRACT() does not support the TIMEZONE_HOUR or TIMEZONE_MINUTE clauses.

Updateable queries

Related topics

These restrictions apply to updates:

- Linking fields cannot be updated
- Index switching will cause an error

Restrictions on live queries

Related topics

Single-table queries are updatable provided that

- There are no JOINs, UNIONs, INTERSECTs, or MINUS operations.
- There is no DISTINCT key word in the SELECT. (This restriction may be relaxed if all the fields of a unique index are projected.)
- Everything in the SELECT clause is a simple column reference or a calculated field; no aggregation is allowed.
- The table referenced in the FROM clause is either an updatable base table or an updatable view.
- There is no GROUP BY or HAVING clause.
- There are no subqueries that reference the table in the FROM clause and no correlated subqueries.
- Any ORDER BY clause can be satisfied with an index.

Restrictions on live joins

Related topics

Live joins may be used only if

- All joins are left-to-right outer joins.
- All join are equi-joins.
- All join conditions are satisfied by indexes.
- Output ordering is not defined.
- The query contains no elements listed above that would prevent single-table updatability.

Constraints

Related topics

You can constrain any updatable query by setting the Query object's *constrained* property to true before activating the query. This causes the query to behave more like an SQL-server-based query. New or modified rows that do not match the conditions of the query will disappear from the result set, although the data is saved.

Statements supported

Related topics

Local SQL supports the following DDL and DML statements:

ALTER TABLE

CREATE INDEX

CREATE TABLE

DELETE

DROP INDEX

DROP TABLE

INSERT

SELECT

FROM clause

WHERE clause

ORDER BY clause

GROUP BY clause

HAVING clause

UNION clause

UPDATE

ALTER TABLE

Related topics

Adds or drops (deletes) one or more columns (fields) from a table.

Syntax

```
ALTER TABLE table
    ADD <column name> <data type> |
    DROP <column name>
    [, ADD <column name> <data type> |
    DROP <column name> ...]
```

Description

Use ALTER TABLE to modify the structure of an existing table. ALTER TABLE with the ADD clause adds the column <*column name*> of the type <*data type*> to <*table name*>. Use the DROP clause to remove the existing column <*column name*> from <*table*>.

Multiple columns may be added and/or dropped in a single ALTER TABLE command.

Use ALTER TABLE as a means of modifying the structure of a table without using the Table Designer.

ALTER TABLE examples

The following statement adds a column:

```
ALTER TABLE "employee.dbf" ADD BUILDING_NO SMALLINT
```

The next statement drops two columns:

```
ALTER TABLE "employee.db" DROP LAST_NAME, DROP FIRST_NAME
```

The following statement drops two columns and adds one:

```
ALTER TABLE "employee.dbf" DROP LAST_NAME, DROP FIRST_NAME, ADD FULL_NAME
CHAR[30]
```

CREATE INDEX

Related topics

Creates a new index on a table.

Syntax

CREATE INDEX <*index name*> ON <*table name*> (<*column name*> [, <*column name*>...])

Description

Use CREATE INDEX to create a new index <*index name*>, in ascending order, based on the values in one or more columns <*column name*> of <*table name*>. Expressions cannot be used to create an index, only columns.

When working with DBF tables, the index can only be created for a single column. The new index is created as a new index tag in the production index. A production index is created if it does not exist. Using CREATE INDEX is the only way to create indexes for DBF tables in SQL.

CREATE INDEX can create only secondary indexes for Paradox tables. Primary Paradox indexes can be created only by specifying a PRIMARY KEY constraint when creating a new table with CREATE TABLE. The secondary indexes are created as case-insensitive and maintained, when possible.

CREATE INDEX examples

The following statement creates an index on a DBF table:

```
CREATE INDEX NAMEX ON employee.dbf (LAST_NAME)
```

The following statement adds an index called ZIP on the ZIP_POSTAL column of the CUSTOMER table:

```
CREATE INDEX ZIP ON CUSTOMER (ZIP_POSTAL)
```

CREATE TABLE

Related topics

Creates a new table.

Syntax

CREATE TABLE <table name> (<*column name*> <*data type*> [,<*column name*> <*data type*>...])

Description

Create a Paradox or dBASE table using local SQL by specifying the file extension when naming the table:

- DB for Paradox tables
- DBF for dBASE tables

If you omit the file extension for a local table name, the table created is the table type specified in the Default Driver setting in the System page of the BDE Configuration Utility.

At least one <*column name*> <*data type*> must be defined. The column definition list must be enclosed in parentheses.

CREATE TABLE is a alternate way of creating a table without using the Table Designer, the Database object's copyTable() method, or an UpdateSet object.

Data type mappings for CREATE TABLE

Related topics

The following table lists SQL syntax for data types used with CREATE TABLE, and describes how those types are mapped to Paradox and dBASE types by BDE:

SQL syntax	Paradox	dBASE
SMALLINT	Short	Number (6,10)
INTEGER	Long Integer	Number (20,4)
DECIMAL(x,y)	BCD	N/A
NUMERIC(x,y)	Number	Number (x,y)
FLOAT(x,y)	Number	Float (x,y)
CHARACTER(n)	Alpha	Character
VARCHAR(n)	Alpha	Character
DATE	Date	Date
BOOLEAN	Logical	Logical
BLOB(n,1)	Memo	Memo
BLOB(n,2)	Binary	Binary
BLOB(n,3)	Formatted memo	N/A
BLOB(n,4)	OLE	OLE
BLOB(n,5)	Graphic	N/A
TIME	Time	N/A
TIMESTAMP	Timestamp	N/A
MONEY	Money	Number (20,4)
AUTOINC	Autoincrement	N/A
BYTES(n)	Bytes	N/A

x = precision (default: specific to driver)

y = scale (default: 0)

n = length in bytes (default: 0)

1–5 = BLOB subtype (default: 1)

CREATE TABLE examples

The following example creates a DBF table called SALES with the following structure:

SALES.DBF structure

Field name	Field type	Field length	Decimal places
SALESID	Character	6	
CUSTOMERID	Character	10	
ORDERDATE	Date	8	
ORDERNMBR	Numeric	7	0
ORDERAMT	Numeric	9	2
DELIVERED	Logical	1	

```
CREATE TABLE SALES (
   SALESID         CHAR(6),
   CUSTOMERID      CHAR(10),
   ORDERDATE       DATE,
   ORDERNMBR       NUMERIC(7,0),
   ORDERAMT        NUMERIC(9,2),
   DELIVERED       BOOLEAN)
```

The following statement creates a Paradox table with a PRIMARY KEY constraint on the LAST_NAME and FIRST_NAME columns:

```
CREATE TABLE "employee.db" (
   LAST_NAME       CHAR(20),
   FIRST_NAME      CHAR(15),
   SALARY          NUMERIC(10,2),
   DEPT_NO         SMALLINT,
   PRIMARY KEY(LAST_NAME, FIRST_NAME) )
```

The same statement for a dBASE table should omit the PRIMARY KEY definition:

```
CREATE TABLE "employee.dbf" (
   LAST_NAME       CHAR(20),
   FIRST_NAME      CHAR(15),
   SALARY          NUMERIC(10,2),
   DEPT_NO         SMALLINT)
```

DELETE

Related topics

Deletes rows (records) from a table.

Syntax

DELETE FROM <*table name*> [WHERE <*search condition*>]

Description

Use DELETE to delete rows, or records, from <*table name*>. Without the WHERE clause, all the rows in the table are deleted. Use the WHERE clause to specify a <*search condition*>. Only records matching the <*search condition*> are deleted.

When DELETE is run against DBF tables the following rules apply:

1. If a WHERE clause is used, DELETE only **marks** rows for deletion, even if all the rows match the <*search condition*>. In this way, local SQL's DELETE behaves like the dBASE DELETE command. The rows are recallable unless the table is packed.
2. Without the WHERE clause, all the rows in the table are actually deleted. In this case, DELETE behaves like the dBASE ZAP command. The rows are not recallable, and the table will have zero rows.

When DELETE is run against a Paradox table, all the rows matching the <*search condition*> are actually deleted. If no WHERE clause is used, all the rows in the table are deleted. The data in the deleted rows in **not** recallable.

DELETE example

The following example deletes all the rows in a DBF table called CUSTOMER and results in a table with zero rows:

```
DELETE FROM CUSTOMER.DBF
```

The following example marks all the rows in a DBF table called CUSTOMER for deletion, but does not actually delete the rows from the table:

```
DELETE FROM CUSTOMER.DBF WHERE CUSTOMER_N > 0
```

The following example marks all the rows where the CITY field is equal to “Freeport” for deletion in a DBF table called CUSTOMER:

```
DELETE FROM CUSTOMER.DBF WHERE CITY = "Freeport"
```

The following example deletes all the rows where the CITY field is equal to “Freeport” in a Paradox table called CUSTOMER:

```
DELETE FROM CUSTOMER.DB WHERE CITY = "Freeport"
```

DROP INDEX

Related topics

Drops (deletes) an existing index from a table.

Syntax

DROP INDEX <table_name>.<index_name> | PRIMARY

Description

Use DROP INDEX to drop, or delete, the index <*index name*> from <*table name*>. For DBF tables <*index name*> must be the name of a tag in the production index.

The PRIMARY keyword is used to delete a primary Paradox index. For example, the following statement drops the primary index on EMPLOYEE.DB:

```
DROP INDEX "employee.db".PRIMARY
```

To drop any dBASE index, or to drop secondary Paradox indexes, provide the index name. For example, this statement drops a secondary index on a Paradox table:

```
DROP INDEX "employee.db".NAMEX
```

DROP INDEX example

The following statement drops the index tag NAME from the production index of a dBASE table called EMPLOYEE:

```
DROP INDEX EMPLOYEE.NAME
```

DROP TABLE

Related topics

Drops (deletes) a table.

Syntax

DROP TABLE <*table name*>

Description

Use DROP TABLE to delete the table <*table name*> from disk. The associated production index file and memo file, if any, are also deleted.

DROP TABLE example

The following statement drops a table named EMPLOYEE:

```
DROP TABLE EMPLOYEE
```

INSERT

Related topics

Adds new rows (records) to a table.

Syntax

INSERT INTO <*table name*>
[(<*column list*>)] VALUES (<*value list*>) |
SELECT <*command*>

Description

Use INSERT to add rows, or records, to a table. There are two forms of this command. In the first form, you use <*value list*> to specify individual column values that are to be inserted for the new row. The values to be inserted must match in number, order, and type with the columns specified in <*column list*>, if <*column list*> is specified. Columns in the new row for which no value is given are left blank. If no <*column list*> is given, the order of the columns as they appear in the table is assumed. Without a <*column list*> a value must be provided for each column in the <*value list*>.

In the second form, the SELECT clause is executed just like a SELECT command. The row or rows returned by the SELECT are inserted into <*table name*>. The columns of the rows returned by the SELECT are matched up with the columns listed in <*column list*>. Therefore, the columns returned by SELECT must match in number, order, and type with the columns specified in <*column list*>, if <*column list*> is specified. If no <*column list*> is given, the number, order, and type of the columns returned by the SELECT must match the number, order, and type of the columns in <*table name*>.

INSERT examples

The following statement adds a row to a table, assigning values to two columns:

```
INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID) VALUES (52, "DGPII");
```

The next statement specifies values to insert into a table with a SELECT statement:

```
INSERT INTO PROJECTS
   SELECT * FROM NEW_PROJECTS
      WHERE NEW_PROJECTS.START_DATE > '06/06/94';
```

SELECT

Related topics

Retrieves data from one or more tables.

Syntax

```
SELECT [DISTINCT] <column list>
  FROM <table reference>
  [WHERE <search condition>]
  [ORDER BY <order list>]
  [GROUP BY <group list>]
  [HAVING <having condition>]
  [UNION <select expr>]
```

Description

Use SELECT to retrieve data from a table or set of tables based on some criteria.

A SELECT that retrieves data from multiple tables is called a join.

<*column list*> is a comma-delimited list of columns in the table(s) you want to retrieve. The columns are retrieved in the order given in the list. If two or more tables used by SELECT use the same field names, distinguish the tables by using the table name and a dot (.). For example, if you're SELECTing from the CUSTOMER table and the PRODUCT table, and they both have a field called NAME, enter the fields as CUSTOMER.NAME and PRODUCT.NAME in <*column list*>. To retrieve all the columns from <*table list*>, use an asterisk (*) for <*column list*>. To eliminate rows containing duplicate values within the same column, precede the <*column list*> with the keyword DISTINCT.

The <*column list*> indicates the columns from which to retrieve data. For example, the following statement retrieves data from two columns:

```
SELECT PART_NO, PART_NAME
   FROM PARTS
```

A SELECT statement that contains a join must have a WHERE clause in which at least one field from each table is involved in an equality check.

FROM clause

Related topics

The FROM clause specifies the table or tables from which to retrieve data. <table reference> can be a single table, a comma-delimited list of tables, or can be an inner or outer join as specified in the SQL-92 standard. For example, the following statement specifies a single table:

```
SELECT PART_NO FROM PARTS
```

The next statement specifies a left outer join for table_reference:

```
SELECT * FROM PARTS LEFT OUTER JOIN INVENTORY
   ON PARTS.PART_NO = INVENTORY.PART_NO
```

WHERE clause

Related topics

The optional WHERE clause reduces the number of rows returned by a query to those that match the criteria specified in <search condition>. For example, the following statement retrieves only those rows with PART_NO greater than 543:

```
SELECT * FROM PARTS
   WHERE PART_NO > 543
```

In addition to scalar comparison operators (=, <, > ...) additional predicates using IN, LIKE, ANY, ALL, and EXISTS are supported.

The IN predicate is followed by a list of values in parentheses. For example, the next statement retrieves only those rows where a part number matches an item in the IN predicate list:

```
SELECT * FROM PARTS
   WHERE PART_NO IN (543, 544, 546, 547)
```

ORDER BY clause

Related topics

The ORDER BY clause specifies the order of retrieved rows, using the keywords ASC (the default) and DESC for ascending and descending, respectively. For example, the following query retrieves a list of all parts listed in alphabetical order by part name:

```
SELECT * FROM PARTS
   ORDER BY PART_NAME ASC
```

The next query retrieves all part information ordered in descending numeric order by part number:

```
SELECT * FROM PARTS
   ORDER BY PART_NO DESC
```

Calculated fields can be ordered by correlation name or ordinal position. For example, the following query orders rows by FULL_NAME, a calculated field:

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME, PHONE
   FROM CUSTOMER
   ORDER BY FULL_NAME
```

Projection of all grouping or ordering columns is not required.

GROUP BY clause

Related topics

The GROUP BY clause specifies how retrieved rows are grouped for aggregate functions. For example,

```
SELECT PART_NO, SUM(QUANTITY) AS PQTY
   FROM PARTS
   GROUP BY PART_NO
```

Aggregates in the SELECT clause must have a GROUP BY clause if a projected field is used, as shown in the example above.

HAVING clause

Related topics

The HAVING clause specifies conditions records must meet to be included in the return from a query. It is a conditional expression used in conjunction with the GROUP BY clause. Groups that do not meet the expression in the HAVING clause are omitted from the result set.

Subqueries are supported in the HAVING clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or parent, query. See WHERE Clause.

In addition to scalar comparison operators (=, <, > ...) additional predicates using IN, LIKE, ANY, ALL, and EXISTS are supported.

UNION clause

Related topics

The UNION clause combines the results of two or more SELECT statements to produce a single table.

Heterogeneous joins

Related topics

Local SQL supports joins of tables in different database formats; such a join is called a heterogeneous join.

When you specify a table name after selecting a local alias,

- For local tables, specify either the alias or the path.
- For remote tables, specify the alias.

The following statement retrieves data from a Paradox table and a dBASE table:

```
SELECT DISTINCT C.CUST_NO, C.STATE, O.ORDER_NO
   FROM CUSTOMER.DB C, ORDER.DBF O
   WHERE C.CUST_NO = O.CUST_NO
```

You can also use BDE aliases in conjunction with table names.

Heterogeneous joins examples

The following is a basic query that selects an entire table:

```
SELECT * FROM BIOLIFE
```

The following examples show simple SELECTs:

```
SELECT NAME, PHONE FROM CUSTOMER WHERE STATE_PROV = "CA"
SELECT CUSTOMER_NO FROM CUSTOMER WHERE LAST_NAME = "Johnson"
SELECT PART_NO, SUM(QUANTITY) AS PQTY FROM PARTS GROUP BY PART_NO
```

The following example illustrates the ORDER BY with a DESCENDING clause:

```
SELECT DISTINT CUSTOMER_NO
   FROM "C:/DATA/CUSTOMER"
   ORDER BY CUSTOMER_NO DESCENDING
```

The following example illustrates how the SELECT statement is supported as an equivalent to a JOIN:

```
SELECT DISTINCT P.PART_NO, P.QUANTITY, G.CITY
   FROM PARTS P, GOODS G
   WHERE P.PART_NO = G.PART_NO
   AND P.QUANTITY > 20
   ORDER BY P.QUANTITY, G.CITY, P.PART_NO
```

Sub-select queries are supported. The following example illustrates this syntax:

```
SELECT P.PART_NO
   FROM PARTS P
   WHERE P.QUANTITY IN
      (SELECT I.QUANTITY
         FROM INVENTORY I
         WHERE I.PART_NO = 'AA9393')
```

The following example shows a join in which fields from each table are involved in some type of equality check and require a WHERE clause:

```
SELECT DISTINCT PARTS.PART_NO, PARTS.QUANTITY, GOODS.CITY
   FROM PARTS, GOODS
   WHERE PARTS.PART_NO = GOODS.PART_NO AND PARTS.QUANTITY > 20
   ORDER BY PARTS.QUANTITY, GOODS.CITY, PARTS.PART_NO
```

The following example shows the use of the DESCENDING keyword in the ORDER BY clause. Note that in this case you must also specify DISTINCT.

```
SELECT DISTINCT CUSTOMER_NO
   FROM CUSTOMER
   ORDER BY CUSTOMER_NO DESCENDING
```

UPDATE

Related topics

Adds or changes values in existing columns in existing rows of a table.

Syntax

UPDATE <*table name*>
  SET <*column name*> = <*expression*> [, <*column name*> = <*expression*>...]
  WHERE <*search condition*>

Description

Use UPDATE to update (change) values within existing columns in existing rows of a table. The column specified by <*column name*> is updated with the value of <*expression*> in all rows that match the <*search criteria*> of the WHERE clause. If the WHERE clause is omitted, the column is updated in all rows in the table. Multiple columns may be updated in a single UPDATE command. A given column of a table may only appear once to the left of an equal sign (=) in the SET clause.

~HEAD ^# IDH_LSQL_UPDATE_EX ^$ UPDATE example ^> ex_win ^C3 HEAD~>UPDATE example

Data access objects

Data access objects provide access to database tables and are used to link tables to the user interface.

The Borland Database Engine (BDE) considers two table types to be Standard tables: DBF (dBASE) and DB (Paradox). The BDE can access any Standard table directly through its path and file name, without having to use a BDE alias.

All other table types, including InterBase, Oracle, Microsoft/Sybase SQL Server, Informix, and any ODBC connection, require the creation of a BDE alias through the BDE Configuration Utility. You may also create a BDE alias to access Standard tables. In that case, the alias specifies the directory in which the tables exist; the database consists of the Standard tables in that directory, and you may not open any others from another directory.

All tables, whether or not they require a BDE alias, are accessed through SQL and the data access objects.

NEXT...

Data access objects: Class hierarchy

To understand the implications of using a BDE alias, you need to understand the class hierarchy of the data access objects.

At the top of the hierarchy is IntraBuilder itself. Next is the *Session* class. A session represents a separate user task, and is required primarily for DBF and DB table security. Multiple sessions may be due to multiple users, a single user doing different things, or both. For example, one user might be accessing an encrypted payroll table with full rights, and looking at a vacation request form; this would be two sessions. A second user could be attached to the same IntraBuilder Agent and looking at the payroll table with read-only rights, which is another session. With those two users, the IntraBuilder Agent would be supporting three sessions. The IntraBuilder Designer and each IntraBuilder Agent supports up to 32 simultaneous sessions, but each session may support an unlimited number of users as long as they don't require their own session-based services. When IntraBuilder first starts, it already has a default session.

Each session contains one or more Database objects. A session always contains a default Database object, one that has no BDE alias and is intended to directly access Standard tables. You must create new Database objects to use tables through a BDE alias. Once you set the BDE alias, activate the Database object, and log in if necessary, you have access to that database's tables. You may also log transactions or buffer updates to each database to allow you to rollback, abandon, or post changes as desired.

Accessing tables

The *Query* object acts primarily as a container for an SQL statement and the set of rows, or *rowset*, that results from it. A rowset represents all or part of a single table or group of related tables. There is only one rowset per query, but you may have more than one query, and therefore more than one rowset, per database. A rowset maintains the current record or row, and therefore contains the typical navigation, buffering, and filtering methods.

The SQL statement may also contain parameters, which are represented in the Query object's *params* array.

Finally, a rowset also contains a *fields* property, which is an array of field objects that contain information about the fields and the values of the fields for the current row. There are events that allow you to morph the values so that the values stored in the table are different than the values displayed. Each field object can also be linked to a visual component through the component's *dataLink* property to form a link between the user interface and the table. When the two objects are linked in this way, they are said to be *dataLink*ed.

Putting the data access objects together

If you're using Standard tables only, at the minimum you create a query, which gets assigned to the default database in the default session, set the SQL statement and make the query active. If the query is successful, it generates a rowset, and you can access the data through the *fields* array.

When accessing tables through a BDE alias, you will need to create a new database, create the query, assign the database to the query, then set the SQL and make the query active.

If you use the Form Designer or Report Designer, you design these relationships visually and code is generated.

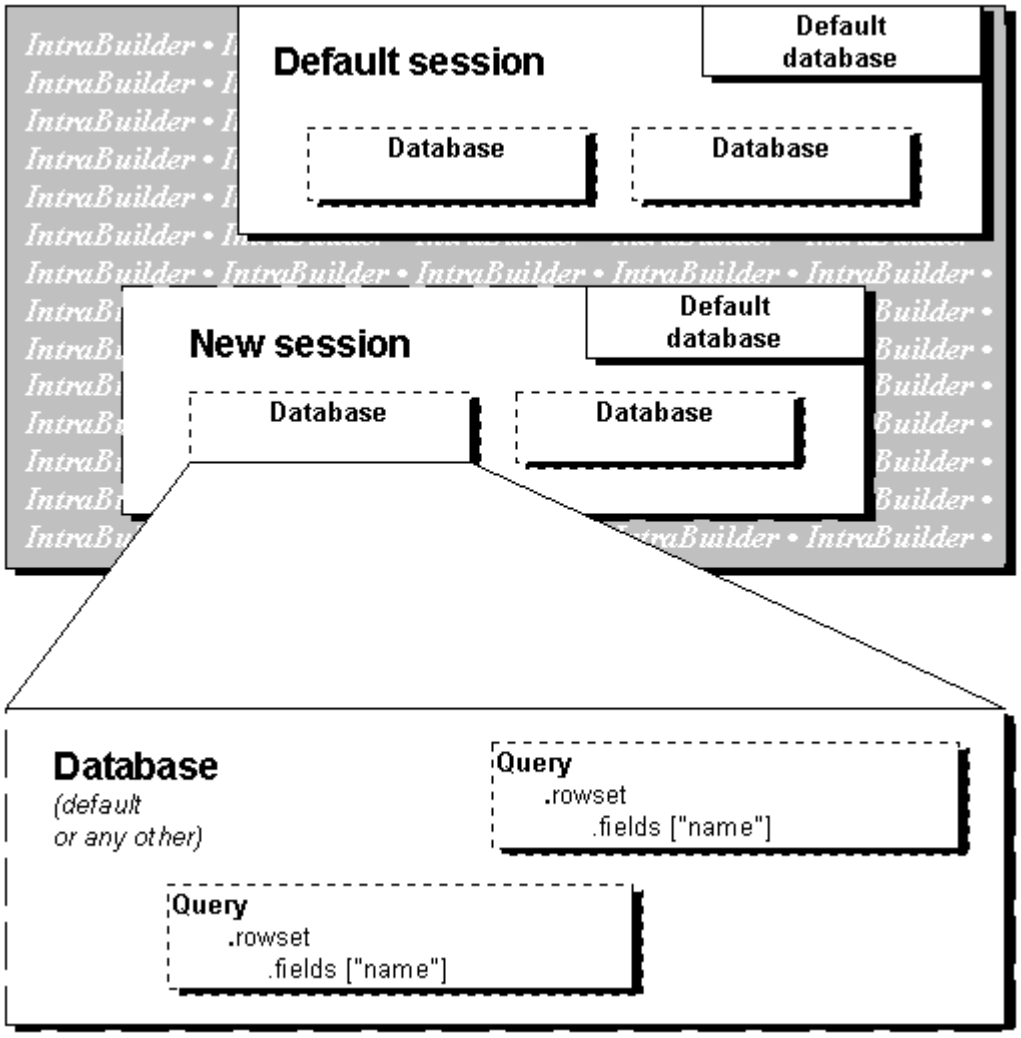
Using stored procedures

The object hierarchy for using stored procedures in an SQL-server database is very similar to the one used for accessing tables. The difference is that a StoredProc object is used instead of a Query object. Above the StoredProc object, the Database and Session objects do the same thing. If the stored procedure returns a rowset, the StoredProc object contains a rowset, just like a Query object.

A StoredProc object also has a *params* array, but instead of simple values to substitute into an SQL statement in a Query object, the *params* array of a StoredProc object contains Parameter objects. Each object describes both the type of parameter—input, output, or result—and the value of that parameter.

Before running the stored procedure, input values are set. After the stored procedure runs, output and result values can be read from the *params* array, or data can be accessed through its rowset.

Data access objects: Class hierarchy diagram



class CalcField

Related topics Example

A calculated field.

Syntax

[<oRef> =] new CalcField(<name expC>)

<oRef>

A variable or property in which to store the reference to the newly created CalcField object.

<name expC>

The name of the calculated field. It cannot duplicate the name of any other field in the query.

Properties

The following tables list the properties and events of the CalcField class. (No methods are associated with this class.) For details on each property, click on the property below.

Property	Default	Description
className	CalcField	Identifies the object as an instance of the CalcField class
fieldName		Name of the object
length	0	Maximum length
parent	null	*fields* array that contains the object
type	Character	Identifies calculated field's return type
value	Empty string	Current value of calculated field

Event	Parameters	Description
beforeGetValue		When *value* property is to be read; return value is used as *value*
onGotValuebeforeGetValue		After *value* is read

Description

Use a calculated field to generate a value based on one or more fields, or some other calculation. For example, in a line item table with both the quantity ordered and price per item, you can calculate the total price for that line item. There would be no need to actually store that total in the table, which wastes space.

Because a calculated field is treated like a field in most respects, you can do things like *dataLink* it to a control on a form or use it in a report. Since a calculated field does not actually represent a field in a table, writing to its value property directly or changing its value through a *dataLink*ed control never causes a change in a table.

After creating a CalcField object, you must add it to the *fields* array of a Rowset object. Because a rowset is not valid until its query opens, you must make the query active before you add the CalcField object. To set the value of a CalcField object, you can do one of two things:

- Assign a code-reference, either a codeblock or function pointer, to the CalcField object's *beforeGetValue* event. The return value of the code becomes the CalcField object's value.
- Assign a value to the CalcField object's *value* property directly as needed, like in the rowset's *onNavigate* event.

class CalcField example

The first example uses the CalcField's *beforeGetValue* event to calculate the total price from the quantity and price per item for each line item:

```
q = new Query();
q.sql = "select * from LINEITEM";
q.active = true;
c = new CalcField( "Total" );
q.rowset.fields.add( c );
c.beforeGetValue = {||this.parent["Quantity"].value *
this.parent["PricePer"].value};
```

Because *this* refers to the CalcField object itself, *this*.*parent* refers to the *fields* array, through which you can access the other field objects. The second example adds a CalcField to an existing *fields* array:

```
q.rowset.fields.add( new CalcField( "Commission" ) );
```

It then uses the following code in the rowset's *onNavigate* event to set the CalcField object's *value* property:

```
this.fields[ "Commission" ].value = this.fields[ "SellPrice" ].value / 10;
```

class Database

Related topics Example

A session's built-in database or a BDE database alias, which gives access to tables.

Syntax

[<oRef> =] new Database()

<oRef>

A variable or property—typically of a Form or Report object—in which to store a reference to the newly created Database object.

Properties

The following tables list the properties and methods of the Database class. (No events are associated with this class.) For details on each property, click on the property below.

Property	Default	Description
active	false	Whether the database is open and active or closed
cacheUpdates	false	Whether to cache changes locally for batch posting later
className	Database	Identifies the object as an instance of the Database class
databaseName	Empty string	BDE alias, or empty string for built-in database
driverName	Empty string	Type (table format or server) of database
handle		BDE database handle
isolationLevel	Read committed	Isolation level of transaction
loginString	Empty string	User name and password to automatically try when opening database
parent	null	Container form or report
session	Default session	Session to which a database is assigned

Method	Parameters	Description
abandonUpdates()		Discards all cached changes
applyUpdates()		Attempts to post cached changes
beginTrans()		Begins transaction; starts logging changes
commit()		Commits changes made during transaction; ends transaction
copyTable()	<source name expC>, <destination name expC>	Makes a copy of a table in the same database
dropTable()	<table name expC>	Deletes table from database
emptyTable()	<table name expC>	Deletes all records from a table
executeSQL()	<expC>	Pass-through SQL statement
packTable()	<table name expC>	Removes deleted records from DBF or DB table and reconsolidates disk usage
reindex()	<table name expC>	Rebuilds indexes for DBF or DB table
renameTable()	<source name expC>, <destination name expC>	Renames table in database
rollback()		Undoes changes made during transaction; ends transaction
tableExists()	<table name expC>	Whether or not specified table exists in database or on disk

Description

All sessions, including the default session you get when you start IntraBuilder, contain a default database, which can access the Standard table types, DBF (dBASE) and DB (Paradox) tables, without

requiring a BDE alias. Whenever you create a Query object, it is initially assigned to the default database in the default session. If you want to use Standard tables in the default session you don't have to do anything with that Query object's *database* or *session* properties. If you want to use a Standard table in another session, for example to use DBF or DB table security, assign that session to the Query object's *session* property, which causes that session's default database to be assigned to that Query object. Default databases are always active; their *active* property has no effect.

You may also set up a BDE alias to access Standard tables. By referring to your Standard tables through a database alias, you can move the tables to a different drive or directory without having to change any paths in your code. All you would have to do is change the path specification for that alias in the BDE Configuration Utility. When using a BDE alias with Standard tables, you are restricted from opening a table in a different directory.

For all non-Standard table types, you will need to set up a BDE alias for the database if you haven't done so already. After creating a new Database object, you may assign it to another session if desired; otherwise it is assigned to the default session. Then you then need to do the following:

- Assign the BDE alias to the *databaseName* property.
- If you need to log in to that database, either set the *loginString* property if you already know the user name and password; or let the login dialog appear. If a login name and password are needed when accessing a database from a browser, IntraBuilder will automatically send a password form to the browser.
- Set the *active* property to *true*. This attempts to open the named database. If it's successful, you now have access to the tables in the database.

Each database, including any default databases, is able to independently support either transaction logging or cached updates. Transaction logging allows changes to be made to tables as usual, but keeps track of those changes. Those changes can then be undone through a *rollback*(), or OK'd with a *commit*(). In contrast, cached updates are not written to the table as they happen, but are cached locally instead. You can then either abandon all the updates or attempt to apply them as a group. If any of the changes fail to post—for a variety of reasons, like locked records or hardware failures—any changes that did take are immediately undone, and the updates remain cached. You can then attempt to solve the problem and reapply the update, or abandon the changes. You may also want to use cached updates to reduce network traffic.

Each non-Standard database is responsible for its own transaction processing, up to whatever isolation level it supports. For Standard tables opened through the default database, if you want simultaneous multiple transactions, you need to create multiple sessions, because each database can support only one active transaction or update cache, and there is only one default database per session.

All Database objects opened by the IntraBuilder Explorer are listed in the *databases* array property of the _sys object. The default database of the default session is _sys.*databases*[0].

A Database object also encapsulates a number of table maintenance methods. These methods occur in the context of the specified Database object. For example, the *copyTable*() method makes a copy of a table in the same database. To use these methods on Standard tables, call the methods through the default database of the default session; for example,

```
_sys.databases[ 0 ].copyTable( "Stuff", "CopyOfStuff" );
```

class Database example

Suppose you have an Access database named PIBMUG.MDB. You install the ODBC driver for Access and create an alias named PIBMUG in the BDE Configuration Utility. To open that database, execute the following code:

```
d = new Database();
d.databaseName = "PIBMUG";
d.active = true;
```

The second example logs into a database named PERSONNEL in a new session with a preset user name and password:

```
s1 = new Session();
d1 = new Database();
d1.databaseName = "PERSONNEL";
d1.session = s1;
d1.loginString = "visitor/jobsavail";
d1.active = true;
```

class DbError

Related topics

An object that describes a BDE or server error.

Syntax

These objects are created automatically by IntraBuilder when a DbException occurs.

Properties

The following table lists the properties of the DbError class. (No events or methods are associated with this class.) For details on each property, click on the property below.

Property	Default	Description
className	DbError	Identifies the object as an instance of the DbError class
code		BDE error number
context		Field name, table name, and so on, that caused error
message	Empty string	Text to describe the error
nativeCode		Server error code

Description

When an error using a data access object occurs, a DbException is generated. Its *errors* property points to an array of DbError objects.

Each DbError object describes a BDE or SQL server error. If *nativeCode* is zero, the error is a BDE error. If *nativeCode* is non-zero, the error is a server error. The *message* property describes the error.

class DbError example

class DbException

Related topics

An object that describes a data access exception. DbException is an extension of the Exception class.

Syntax

These objects are created automatically by IntraBuilder when an exception occurs.

Properties

The following table lists the properties of the DbException class. DbException objects also contain those properties inherited from the Exception class. (No events or methods are associated with the DbException class.) For details on each property, click on the property below.

Property	Default	Description
className	DbException	Identifies the object as an instance of the DbException class
errors		Array of DbError objects

Description

The DbException class is a subclass of the Exception class. It is generated when an error using a data access object occurs. In addition to the IntraBuilder error code and message, it provides access to BDE and SQL server error codes and messages.

class DbException example

class DbfField

Related topics

A field from a DBF (dBASE) table. DbfField is an extension of the Field class.

Syntax

These objects are created automatically by the rowset.

Properties

The following table lists the properties of the DbfField class. DbfField objects also contain those properties inherited from the Field class. (No events or methods are associated with the DbfField class.) For details on each property, click on the property below.

Property	Default	Description
className	DbfField	Identifies the object as an instance of the DbfField class
decimalLength	0	Number of decimal places if the field is a numeric field
readOnly	false	Specifies whether the field has read-only access

Description

The DbfField class is a subclass of the Field class. It represents a field from a DBF (dBASE) table, and contains properties that are specific to fields of that table type. Otherwise it is considered to be a Field object.

class DbfField example

class Field

Related topics

A base class object that represents a field from a table.

Syntax

These objects are created automatically by the rowset.

Properties

The following tables list the properties, events. and methods of the Field class. For details on each property, click on the property below.

Property	Default	Description
className	Field	Identifies the object as an instance of the Field class
fieldName		Name of the field the Field object represents
length		Maximum length
parent	null	*fields* array that contains the object
type	Character	The field's data type
value	Empty string	Represents current value of field in row buffer

Event	Parameters	Description
beforeGetValue		When value property is to be read; return value is used as value
canChange	<new value>	When attempting to change value property; return value allows or disallows change
onChange		After value property is successfully changed
onGotValue		After *value* is read

Method	Parameters	Description
copyToFile()	<filename expC>	Copies data from BLOB field to external file
replaceFromFile()	<filename expC> [, <append expL>]	Copies data from external file to BLOB field

Description

The Field class acts as the base class for the DbfField (dBASE), PdxField (Paradox), and SqlField (everything else) classes. It contains the properties common to all field types. Each subclass contains the properties specific to that table type.

Each rowset has a *fields* property, which points to an array. Each element of that array is an object of one of the subclasses of the Field class, depending on the table type or types contained in the rowset.

While the *fieldName*, *length*, and *type* properties describe the field and are the same from row to row, the *value* property is the link to the field's value in the table. The *value* property's value reflects the current value of that field for the current row in the row buffer; assigning a value to the *value* property assigns that value to the row buffer. The buffer is not written to disk unless the rowset's *save*() method is called, there is some navigation in the rowset, or the rowset is closed. You can abandon any changes you make to the row buffer by calling the rowset's *abandon*() method.

You may assign a Field object to the *dataLink* property of a control on a form. This makes the control data-aware, and causes it to display the current value of the Field object's *value* property; if changes are made to the control, the new value is written to the Field object's *value* property.

class Field example

class Parameter

Related topics Example

A parameter for a stored procedure.

Syntax

These objects are created automatically by the stored procedure.

Properties

The following table lists the properties of the Parameter class. (No events or methods are associated with this class.) For details on each property, click on the property below.

Property	Default	Description
className	Parameter	Identifies the object as an instance of the Parameter class
type	Input	The parameter type (0=Input, 1=Output, 2=InputOutput, 3=Result)
value		The value of the parameter

Description

Parameter objects represent parameters to stored procedures. Each element of the *params* array of a StoredProc object is a Parameter object. The Parameter objects are automatically created when the *procedureName* property is set, either by getting the parameter names for that stored procedure from the SQL server or by using parameter names specified directly in the *procedureName* property.

A parameter may be one of four types, as indicated by its *type* property:

- Input: an input value for the stored procedure. The *value* must be set before the stored procedure is called.
- Output: an output value from the stored procedure. The *value* must be set to the correct data type before the stored procedure is called; any dummy value may be used. Calling the stored procedure sets the *value* property to the output value.
- InputOutput: both input and output. The *value* must be set before the stored procedure is called. Calling the stored procedure updates the *value* property with the output value.
- Result: the result value of the stored procedure. In this case, the stored procedure acts like a function, returning a single result value, instead of updating parameters that are passed to it. Otherwise, the *value* is treated like an output value. The name of the Result parameter is always “Result”.

A Parameter object may be assigned as the *dataLink* of a component in a form. Changes to the component are reflected in the *value* property of the Parameter object, and updates to the *value* property of the Parameter object are displayed in the component.

class Parameter example

The following statements call a stored procedure that returns an output parameter. The result is displayed in the Script Pad.

```
d = new Database();
d.databaseName = "IBLOCAL";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "DEPT_BUDGET";
p.params[ "DNO" ].value = "670";                    // Set input parameter
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

The following statement calls a stored procedure in a database that does not return any parameter information. Therefore, the parameters must be declared in the *procedureName* property. Note that the parameter names are case-sensitive, and you must initialize any output parameters by assigning a dummy value of the correct data type.

```
#define PARAMETER_TYPE_INPUT        0
#define PARAMETER_TYPE_OUTPUT       1
#define PARAMETER_TYPE_INPUT_OUTPUT 2
#define PARAMETER_TYPE_RESULT       3
d = new Database();
d.databaseName = "WIDGETS";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "PROJECT_SALES( :month, :units )";
p.params[ "month" ].type  = PARAMETER_TYPE_INPUT;
p.params[ "month" ].value = 6;
p.params[ "units" ].type  = PARAMETER_TYPE_OUTPUT;
p.params[ "units" ].value = 0;                      // Output will be numeric
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

class PdxField

Related topics

A field from a DB (Paradox) table. PdxField is an extension of the Field class.

Syntax

These objects are created automatically by the rowset.

Properties

The following table lists the properties of the PdxField class. PdxField objects also contain those properties inherited from the Field class. (No events or methods are associated with the PdxField class.) For details on each property, click on the property below.

Property	Default	Description
className	PdxField	Identifies the object as an instance of the PdxField class
lookupTable	Empty string	Table to use for lookup value
lookupType	Empty string	Type of lookup
maximum		Maximum allowed value for field
minimum		Minimum allowed value for field
picture	Empty string	Formatting template
required	false	Whether the field must be filled in
readOnly	false	Whether the field has read-only access

Description

This class is called PdxField—not “DbField”—to avoid confusion and simple typographical errors between it and the DbfField class.

The PdxField class is a subclass of the Field class. It represents a field from a DB (Paradox) table, and contains properties that are specific to fields of that table type. Otherwise it is considered to be a Field object.

class PdxField example

class Query

Related topics Example

A representation of an SQL statement that describes a query and contains the resulting rowset.

Syntax

[<oRef> =] new Query()

<oRef>

A variable or property—typically of a Form or Report object—in which to store a reference to the newly created Query object.

Properties

The following tables list the properties, events, and methods of the Query class. For details on each property, click on the property below:

Property	Default	Description
active	false	Whether the query is open and active or closed
className	Query	Identifies the object as an instance of the Query class
constrained	false	Whether the WHERE clause of the SQL SELECT statement will be enforced when attempting to update Standard tables
database	null	Database to which the query is assigned
handle		BDE statement handle
masterSource	null	Query that acts as master query and provides parameter values
params	AssocArray	Associative array that contains parameter names and values for the SQL statement
parent	null	Container form or report
requestLive	true	Whether you want a writable rowset
rowset	null	Results of the query
session	null	Session to which the query is assigned
sql	Empty string	SQL statement that describes the query
unidirectional	false	Whether to assume forward-only navigation to increase performance on SQL-based servers
updateWhere	AllFields	Enum to determine which fields to use in constructing the WHERE clause of an SQL UPDATE statement, used for posting changes to SQL-based servers

Event	Parameters	Description
canClose		When attempting to close query; return value allows or disallows closure
canOpen		When attempting to open query; return value allows or disallows opening
onClose		After query closes
onOpen		After query first opens

Method	Parameters	Description
prepare()		Prepares SQL statement
requery()		Rebinds and executes SQL statement

Description

The Query object is where you specify which fields you want from which rows in which tables and the order in which you want to see them, through an SQL SELECT statement stored in the query's *sql* property. The results are accessed through the query's *rowset* property. To use a stored procedure that results in a rowset, use a StoredProc object instead.

Whenever you create a query object, it is initially assigned to the default database in the default session. If you want to use Standard tables in the default session you don't have to do anything with that query's *database* or *session* properties. If you want to use a Standard table in another session, assign that session to the query's *session* property, which causes that session's default database to be assigned to that query.

For non-Standard tables, you will need to set up a BDE alias for the database if you haven't done so already. After creating a new Database object, you may assign it to another session if desired; otherwise it is assigned to the default session. Once the Database object is active, you can assign it to the query's *database* property. If the database is assigned to another session, you need to assign that session to the query's *session* property first.

After the newly created query is assigned to the desired database, an SQL SELECT statement describing the data you want is assigned to the query's *sql* property.

If the SQL statement contains parameters, the Query object's *params* array is automatically populated with the corresponding elements. The value of each array element must be set before the query is activated. A Query with parameters can be used as a detail query in a master-detail relationship through the *masterSource* property.

Setting the Query object's *active* property to *true* opens the query and executes the SQL statement stored in the *sql* property. If the SQL statement fails, for example the statement is misspelled or the named table is missing, an error is generated and the *active* property remains *false*. If the SQL statement executes but does not generate any rows, the *active* property is *true* and the *endOfSet* property of the query's *rowset* is *true*. Otherwise the *endOfSet* property is *false*, and the rowset contains the resulting rows.

Setting the *active* property to *false* closes the query, writing any buffered changes.

class Query example

The first example opens a table named VACATION.DBF:

```
q= new Query();
q.sql = "select * from VACATION";
q.active = true;
```

The second example opens a table named REQS in a database named PERSONNEL in a new session with a preset user name and password:

```
s1 = new Session();
d1 = new Database();
d1.databaseName = "PERSONNEL";
d1.session = s1;
d1.loginString = "visitor/jobsavail";
d1.active = true;
q1 = new Query();
q1.session = s1;
q1.database = d1;
q1.sql = "select * from REQS";
q1.active = true;
```

The third example uses an SQL statement with parameters. Note that the parameter name is case-sensitive; the name in the *params* array must match the name in the SQL statement:

```
q1 = new Query();
q1.sql = "select * from CUSTOMER where STATE = :state";
q1.params[ "state" ] = "VA";
q1.active = true;
```

class Rowset

Related topics Example

The data that results from an SQL statement in a Query object.

Syntax

These objects are created automatically by the query.

Properties

The following tables list the properties, events, and methods of the Rowset class. For details on each property, click on the property below.

Property	Default	Description
autoEdit	true	Whether the rowset defaults to Edit mode or requires *beginEdit*() to be called
className	Rowset	Identifies the object as an instance of the Rowset class
endOfSet		Whether the row cursor is at either end of the set
fields		Array of field objects in row
filter	Empty string	Filter SQL expression
filterOptions	Match length and case	Enum designating how the filter expression should be applied
handle		BDE cursor handle
indexName	Empty string	Active index tag
live	true	Whether the data can be modified
locateOptions	Match length and case	Enum designating how the locate expression should be applied
masterFields	Empty string	Field list for master-detail link
masterRowset	null	Reference to master Rowset object
modified	false	Whether the row has changed
notifyControls	true	Whether to automatically update *dataLink*ed controls
parent	null	Query object that contains the Rowset object
state	Closed	Enum that describes the mode the rowset is in

Event	Parameters	Description
canAbandon		When *abandon*() is called; return value allows or disallows abandoning of row
canAppend		When *beginAppend*() is called; return value allows or disallows start of append
canDelete		When *delete*() is called; return value allows or disallows deletion
canEdit		When *beginEdit*() is called; return value allows or disallows switch to Edit mode
canGetRow		When attempting to read row; return value acts as an additional filter
canNavigate		When attempting row navigation; return value allows or disallows navigation
canSave		When *save*() is called; return value allows or disallows saving of row
onAbandon		After successful *abandon*()
onAppend		After successful *beginAppend*()
onDelete		After successful *delete*()
onEdit		After successful *beginEdit*()

onNavigate	After rowset navigation
onSave	After successful *save*()

Method	Parameters	Description
abandon()		Abandons pending changes to current row
applyFilter()		Applies filter set during rowset's Filter mode
applyLocate()	[<locate expC>]	Finds first row that matches specified criteria
beginAppend()		Starts append of new row
beginEdit()		Puts rowset in Edit mode, allowing changes to fields
beginFilter()		Puts rowset in Filter mode, allowing entry of filter criteria
beginLocate()		Puts rowset in Locate mode, allowing entry of search criteria
bookmark()		Returns bookmark for current row
clearFilter()		Disables filter created by *applyFilter*() and clears *filter* property
count()		Returns number of rows in rowset, honoring filters
delete()		Deletes current row
first()		Moves row cursor to first row in set
goto()	<bookmark>	Moves row cursor to specified row
last()		Moves row cursor to last row in set
locateNext()	[<rows expN>]	Finds other rows that match search criteria
lockRow()		Locks current row
lockSet()		Locks entire set
next()	[<rows expN>]	Navigates to adjacent rows
refresh()		Refreshes entire rowset
refreshControls()		Refreshes *dataLink*ed controls
refreshRow()		Refreshes current row only
save()		Saves current row
unlock()		Releases locks set by *lockRow*() and *lockSet*()

Description

A Rowset object represents a set of rows that results from a query. It maintains a cursor that points to one of the rows in the set, which is considered the current row, and a buffer to manage the contents of that row. The row cursor may also point outside the set, either before the first row or after the last row, in which case it is considered to be at the end-of-set. Each row contains fields from one or more tables. These fields are represented by an array of Field objects that is represented by the rowset's *fields* property. For a simple query like the following, which selects all the fields from a single table with no conditions,
the rowset represents all the data in the table:

```
select * from CUSTOMER
```

As the cursor moves from row to row, you can access the fields in that row.

A Query object always has a *rowset* property, but that rowset is not open and usable and does not contain any fields until the query has been successfully activated. Setting the Query object's *active* property to *true* opens the query and executes the SQL statement stored in the *sql* property. If the SQL statement fails, for example the statement is misspelled or the named table is missing, an error is generated and the *active* property remains *false*. If the SQL statement executes but does not generate any rows, the *active* property is *true* and the *endOfSet* property of the query's *rowset* is *true*. Otherwise the *endOfSet* property is *false*, and the rowset contains the resulting rows.

Once the rowset has been opened, you can do any of the following:

- Navigate the rowset; that is, move the row cursor

- Filter and search for rows
- Add, modify, and delete rows
- Explicitly lock individual rows or the entire set

The individual Field objects in a rowset's *fields* array property may be *dataLink*ed to controls on a form. As the row cursor is navigated from row to row, the controls will be updated with the current row's values, unless the rowset's *notifyControls* property is set to *false*. Changing the values shown in the controls will change the *value* property of the *dataLink*ed Field objects. You may also directly modify the *value* property of the Field objects. All of the values are maintained in the row buffer.

Rowset objects support master-detail linking. Navigation and updates in the master rowset change the set of rows in the detail rowset. The detail rowset is created by changing the key range of an existing index in the detail rowset. The *masterRowset* and *masterFields* properties are set in the detail rowset. This allows a single master rowset to control any number of detail rowsets.

By default, a rowset's *autoEdit* property is *true*, which means that a rowset opens in Edit mode; its fields are changeable. By setting *autoEdit* to *false*, the rowset in Browse mode, which gives read-only access, and the *beginEdit*() method must be called to switch to Edit mode and allow editing. This is particularly useful because of the remote nature of Web access.

Changes made to the row buffer are not written until either the *save*() method is called, there is navigation in the rowset, or the rowset is closed by deactivating the query. The rowset's *modified* property indicates whether any changes have been made.

In addition to normal data access through Browse and Edit modes, the rowset supports three other modes: Append, Filter, and Locate, which are initiated by *beginAppend*(), *beginFilter*(), and *beginLocate*() respectively. At the beginning
of all three modes, the row buffer is disassociated from whatever row it was buffering and cleared. This allows the entry of field values typed into *dataLink*ed controls or assigned directly to the *value* property. In Append mode, these new values are saved as a new row if the row buffer is written. In Filter mode, executing an *applyFilter*() causes the non-blank field values to be used as criteria for filtering rows, showing only those that match. In Locate mode, calling *applyLocate*() causes the non-blank field values to be used as criteria to search for matching rows. In all three modes, using the field values cancels that mode. Also, calling the *abandon*() method causes the rowset to revert back to Browse or Edit mode without using the values.

You can easily implement filter-by-form and locate-by-form features with the Filter and Locate modes. Instead of using Filter mode, you can assign an SQL expression directly to the rowset's *filter* property. The rowset's *canGetRow* event will filter rows based on any JavaScript code, not just an SQL expression, and can be used instead of or in addition to Filter mode and the *filter* property. You can also use *applyLocate*() without starting Locate mode first by passing an SQL expression to find the first row for which the expression is *true*.

Any row-selection criteria—from the WHERE clause of the query's SQL SELECT statement, the key range enforced by a master-detail link, or a filter—
is actively enforced. *applyLocate*() will not find a row that does not match the criteria. When appending a new row or changing an existing row, if the fields
in the row are written such that the row no longer matches the selection criteria, that row becomes out-of-set, and the row cursor moves to the next row, or to the end-of-set if there are no more matching rows. To see the out-of-set row, you must remove or modify the selection criteria to allow that row.

Row and set locking support varies among different table types. The Standard (DBF and DB) tables fully support locking, as do some SQL servers. For servers that do not support true locks, the Borland Database Engine emulates optimistic locking. Any lock request is assumed to succeed. Later, when the actual attempt to change the data occurs, if the data has changed since the lock attempt, an error occurs.

In the IntraBuilder Designer, any attempt to change the data in a row, like typing a letter in a *dataLink*ed Text control, causes an automatic row lock to be attempted. If that row is already locked, the lock is retried up to the number of times specified by the session's *lockRetryCount* property; if after those attempts the lock is unsuccessful, the change does not take. If the automatic lock is successful, the lock remains until navigation off the locked row occurs; then the lock is automatically removed. In contrast,

because of the detached nature of a client browser, data in controls is always submitted as a group when the form is submitted. Only a momentary lock is required to post the data. In addition, the data may have already been changed between the time the data was read and displayed on the client browser and the time changes are posted.

class Rowset example

The following code gives everyone an extra day of vacation:

```
q= new Query();
q.sql = "select * from EMPLOYEE";
q.active = true;
while ( !q.rowset.endOfSet ) {
  q.rowset.fields[ "VacHours" ].value += 8;
  q.rowset.next();
}
```

class Session

Related topics Example

An object that manages simultaneous database access.

Syntax

[<oRef> =] new Session()

<oRef>

A variable or property—typically of a Form or Report object—in which to store a reference to the newly created Session object.

Properties

The following table lists the properties and methods of the Session class. (No events are associated with this class.) For details on each property, click on the property below.

Property	Default	Description
className	Session	Identifies the object as an instance of the Session class
handle		BDE session handle
lockRetryCount	0	Number of times to retry a failed lock attempt
lockRetryInterval	0	Number of seconds to wait between each lock attempt
parent	null	Container form or report

Method	Parameters	Description
access()		Returns the user's access level for the session
addPassword()	<password expC>	Adds a password to the password table for access to encrypted DB (Paradox) tables
login()	<group expC>, <user expC>, <password expC>	Logs the specified user into the session to access encrypted DBF (dBASE) tables
user()		Returns the user's login name for the session

Description

A session represents a separate user task, and is required primarily for DBF and DB table security. Multiple sessions may be due to multiple users, a single user doing different things, or both. For example, one user might be accessing an encrypted payroll table with full rights, and looking at a vacation request form; this would be two sessions. A second user could be attached to the same IntraBuilder Agent and looking at the payroll table with read-only rights, which is another session. With those two users, the IntraBuilder Agent would be supporting three sessions. The IntraBuilder Designer and each IntraBuilder Agent supports up to 32 simultaneous sessions, but each session may support an unlimited number of users as long as they don't require their own session-based services. When IntraBuilder first starts, it already has a default session.

DBF and DB table security is session-based. (SQL-table security is database-based.) There are a number of different approaches to session-based security. For example, if you decide to give everyone the same level of access, you can use the *login*() or *addPassword*() methods in the default session, and everyone (and all their Query objects) can share the default session. No other Session objects would have to be created. If you decide that everyone must log in by themselves, then they must have their own session. In that case, you would create a Session object on the same form or report that has the Query object, and assign that session to the Query object.

Unlike the Database and Query objects, a Session object does not have an *active* property. Sessions are always active. To close a session, you must destroy it by releasing all references to it.

class Session example

This example assigns a query that accesses the encrypted PAYROLL.DBF table to a new Session object. When the query is activated, a password form will be automatically generated by IntraBuilder to access the table.

```
s1 = new Session();
q1 = new Query();
q1.session = s1;
q1.sql = "select * from PAYROLL";
q1.active = true;
```

If someone else executed the same code on the same IntraBuilder Agent, they would get their own Session object, and would have to log in themselves.

class SqlField

Related topics

A field from an SQL-server-based table. SqlField is an extension of the Field class.

Syntax

These objects are created automatically by the rowset.

Properties

The following table lists the properties of the SqlField class. SqlField objects also contain those inherited from the Field class. (No events or methods are associated with the SqlField class.) For details on each property, click on the property below.

Property	Default	Description
className	SqlField	Identifies the object as an instance of the SqlField class
precision		The number of digits of precision
scale		How the number is scaled

Description

The SqlField class is a subclass of the Field class. It represents a field from an SQL-server-based table, including any ODBC connection, and contains properties that are specific to fields of that table type. Otherwise it is considered to be a Field object.

class SqlField example

class StoredProc

Related topics Example

A representation of a stored procedure call.

Syntax

[<oRef> =] new StoredProc()

<oRef>

A variable or property—typically of a Form or Report object—in which to store a reference to the newly created StoredProc object.

Properties

The following tables list the properties, events, and methods of the StoredProc class. For details on each property, click on the property below:

Property	Default	Description
active	false	Whether the stored procedure is open and active or closed
className	StoredProc	Identifies the object as an instance of the StoredProc class
database	null	Database to which the stored procedure is assigned
handle		BDE statement handle
params	AssocArray	Associative array that contains Parameter objects for the stored procedure call
parent	null	Container form or report
procedureName	Empty string	Name of the stored procedure
rowset	null	Results of the stored procedure call
session	null	Session to which the stored procedure is assigned

Event	Parameters	Description
canClose		When attempting to close stored procedure; return value allows or disallows closure
canOpen		When attempting to open stored procedure; return value allows or disallows opening
onClose		After stored procedure closes
onOpen		After stored procedure first opens

Method	Parameters	Description
prepare()		Prepares stored procedure call
requery()		Rebinds and executes stored procedure

Description

Use a StoredProc object to call a stored procedure in a database. Most stored procedures take one or more parameters as input and may return one or more values as output. Parameters are passed to and from the stored procedure through the StoredProc object's *params* property, which points to an associative array of Parameter Objects.

Some stored procedures return a rowset. In that case, the StoredProc object is similar to a Query object; but instead of executing an SQL statement that describes the data to retrieve, you name a stored procedure, pass parameters to it, and execute it. The resulting rowset is accessed through the StoredProc object's *rowset* property, just like in a Query object.

Because stored procedures are SQL-server-based, you must create and activate a Database object and assign that object to the StoredProc object's *database* property. Standard tables do not support stored procedures.

Next, the *procedureName* property must be set to the name of the stored procedure. For most SQL

servers, the BDE can get the names and types of the parameters for the stored procedure. On some servers, no information is available; in that case you must include the parameter names in the *procedureName* property as well.

Getting or specifying the names of the parameters automatically creates the corresponding elements in the StoredProc object's *params* array. Each element is a Parameter object. Again, for some servers, information on the parameter types is available. For those servers, the *type* properties are automatically filled in and the *value* properties are initialized. For other servers, you must supply the missing *type* information and initialize the *value* to the correct type.

To call the stored procedure, set its *active* property to *true*. If the stored procedure does not generate a rowset, the *active* property is reset to *false* after the stored procedure executes and returns its results, if any. This facilitates calling the stored procedure again if desired, after reading the results from the *params* array.

If the stored procedure generates a rowset, the *active* property remains true, and the resulting rowset acts just like a rowset generated by a Query object.

You can *dataLink* components in a form to fields in a rowset, or to the Parameter objects in the *params* array.

class StoredProc example

The following statements call a stored procedure that returns an output parameter. The result is displayed in the Script Pad.

```
d = new Database();
d.databaseName = "IBLOCAL";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "DEPT_BUDGET";
p.params[ "DNO" ].value = "670";
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

The following statement calls a stored procedure in a database that does not return any parameter information. Therefore, the parameters must be declared in the *procedureName* property. Note that the parameter names are case-sensitive, and you must initialize any output parameters by assigning a dummy value of the correct data type.

```
#define PARAMETER_TYPE_INPUT        0
#define PARAMETER_TYPE_OUTPUT       1
#define PARAMETER_TYPE_INPUT_OUTPUT 2
#define PARAMETER_TYPE_RESULT       3
d = new Database();
d.databaseName = "WIDGETS";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "PROJECT_SALES( :month, :units )";
p.params[ "month" ].type  = PARAMETER_TYPE_INPUT;
p.params[ "month" ].value = 6;
p.params[ "units" ].type  = PARAMETER_TYPE_OUTPUT;
p.params[ "units" ].value = 0;                      // Output will be numeric
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

class UpdateSet

Related topics Example

An object that updates one table with data from another.

Syntax

[<oRef> =] new UpdateSet()

<oRef>

A variable or property in which to store a reference to the newly created UpdateSet object.

Properties

The following tables list the properties and methods of the UpdateSet class. (No events are associated with this class.) For details on each property, click on the property below.

Property	Default	Description
changedTableName		Table to collect copies of original values of changed rows
className	UpdateSet	Identifies the object as an instance of the UpdateSet class
destination		Rowset object or table name that is updated or created
indexName		Name of index to use
keyViolationTableName		Table to collect rows with duplicate primary keys
problemTableName		Table that collects problem rows
source		Rowset object or table name that contains updates

Method	Parameters	Description
append()		Adds new rows
appendUpdate()		Updates existing rows and adds new rows
copy()		Creates destination table
delete()		Deletes rows in destination that match rows in source
update()		Updates existing rows

Description

The UpdateSet object is used to update data from one rowset to another, or to copy or convert data from one format to another, either in the same database or across databases.

To update a DBF table with *appendUpdate*(), *delete*(), or *update*(), the *indexName* property of the UpdateSet object must be set to a valid index. To update a DB table with the same operations, the DB table's key is used by default, or you can assign a secondary index to the *indexName* property.

The *source* and *destination* can be either a character string containing the name of a table, or an object reference to a rowset. If the source is a rowset, the data used in the update can be filtered.

For Standard table names, specify the name of the table and the extension (DBF or DB). For all other tables, place the database name (the BDE alias) in colons before the table name; that is, in this form:

```
:alias:table
```

The named database must be open when the UpdateSet() method is executed.

class UpdateSet example

The following example copies the result set from a query on an SQL-based-server to a local DBF file:

```
d = new Database();
d.databaseName = "SOMESQL";
d.active = true;
q = new Query();
q.database = d;
q.sql = "select * from SOMETABLE where THIS = 'that' order by ID";
q.active = true;
u = new UpdateSet();
u.source = q.rowset;
u.destination = "RESULTS.DBF";
u.copy();
```

This example copies all the rows from the same SQL-based-server table to a local DBF file without using a Query object:

```
d = new Database();
d.databaseName = "SOMESQL";
d.active = true;
u = new UpdateSet();
u.source = ":SOMESQL:SOMETABLE";
u.destination = "SOMEDUP.DBF";
u.copy();
```

abandon()

Related topics Example

Abandons any pending changes to the current row.

Syntax

<oRef>.*abandon*()

<oRef>

The rowset whose current row buffer you want to abandon.

Property of

Rowset

Description

Changes made to a row, either through *dataLink*ed controls or by assigning values to the *value* property of fields, are not written to disk until there is navigation in the rowset, the rowset's *save*() method is called, or the rowset's query is closed. You can discard any pending changes to the rowset with the *abandon*() method. This is usually done in response to the user's request.

You can check the *modified* property first to see if there have been any changes made to the row. Calling *abandon*() when there's nothing to abandon has no ill effects (although the *canAbandon* event is still fired).

You may also want to discard unwritten changes when a query is closed, the opposite of the default behavior. If you are relying on the query's event handlers to do this instead of abandoning and closing the query through code, you must call *abandon*() during the query's *canClose* event and return *true* from the *canClose* event handler; calling *abandon*() during the *onClose* event will have no effect, since the *onClose* event fires after the query has already closed, and any changes have been written.

When using *abandon*() to discard changes to an existing row, all fields are returned to their original values and any *dataLink*ed controls are automatically restored. If the row was automatically locked when editing began, it is unlocked.

You may also use *abandon*() to discard a new row created by the *beginAppend*() method, in which case the new row is discarded, and the row that was current at the time *beginAppend*() was called is restored. *abandon*() also cancels a rowset's Filter or Locate mode in the same manner.

While *abandon*() discards unwritten changes to the current row, there are two mutually exclusive ways of abandoning changes to more than one row in more than one table in a database, which you can use instead of or in addition to single-row buffering. Calling *beginTrans*() starts transaction logging which logs all changes and allows you to undo them by calling *rollback*() if necessary. The alternative is to set the database's *cacheUpdates* property to *true* so that changes are written to a local cache but not written to disk, and then call *abandonUpdates*() to discard all the changes if needed.

abandon() example

The following *onServerClick* event handler for an Abandon button calls the *abandon*() method for the form's primary rowset:

```
function abandonButton_onServerClick()
{
  this.form.rowset.abandon();
}
```

abandonUpdates()

Related topics Example

Abandons all cached updates in the database.

Syntax

<oRef>.*abandonUpdates*()

<oRef>

The database whose cached changes you want to abandon.

Property of

Database

Description

abandonUpdates() discards all changes to a database that have been cached. Unlike *applyUpdates*(), it cannot fail. See *cacheUpdates* for more information on caching updates.

Changes to the current row that have not been written are still in the row buffer, and have not been cached. To abandon changes made to the row buffer, call the rowset's *abandon*() method.

abandonUpdates() example

Suppose you have a form that's used for redeeming prizes for points accumulated for dining at the corporate cafeteria. As each prize is chosen, the choice is written to the prize redemption table, using cached updates. The points aren't actually spent until you press the Redeem button, and you can cancel all the choices that have been made and start over by pressing the Start Over button. The following is the *onServerClick* event handler for the Start Over button.

```
function startOverButton_onServerClick()
{
  this.form.rowset.parent.database.abandonUpdates();  // Discard cached
updates
  this.form.rowset.abandon();                         // and current choice
}
```

access()

Related topics

Returns the access level of the current session for DBF table security.

Syntax

<oRef>.*access*()

<oRef>

The session you want to test.

Property of

Session

Description

DBF table security is session-based. All queries assigned to the same session in their *session* property have the same access level. Access will be assigned in one of two ways: everyone who needs to open an encrypted table will either be assigned to their own session, or they will all share the same session (for example, you might set up a guest account that everyone uses by default).

access() returns the access level for the current session. For people with their own sessions, that will be their individual access level for the group name and login name they used. For a session that is shared by everyone, that will be the shared access level.

access() returns a number from 0 to 8. 8 is the lowest level of access, 1 is the highest level of access, and 0 is returned if the session is not using DBF security.

active

Related topics

Specifies whether an object is open and active or closed.

Property of

Database, Query, StoredProc

Description

When created, a new session's default database is active since it does not require any setup. Other Database objects, Query objects, and StoredProc objects do require setup, so their *active* property defaults to *false*. Once they have been set up, set their *active* property to *true* to open the object and make it active.

When a Query or StoredProc object's *active* property is set to *true*, its *canOpen* event is called. If there is no *canOpen* event handler, or the event handler returns *true*, the object is activated. In a Query object, the SQL statement in its *sql* property is executed; in a StoredProc object, the stored procedure named in its *procedureName* property is called. Then the object's *onOpen* event is fired.

To close the object, set its *active* property to *false*. Closing an object closes all objects below it in the class hierarchy. Attempting to close a Query or StoredProc object calls its *canClose* event. If there is no *canClose* event handler, or the event handler returns *true*, the object is closed. Closing a Database object closes all its Query and StoredProc objects. After the objects are closed, all the Query and StoredProc objects' *onClose* events are fired.

Closing a query or a StoredProc object that generated a rowset attempts to write any changes to its rowset's current row buffer, and to apply all cached updates or commit all logged changes. To circumvent this, you must call the *abandon*(), *abandonUpdates*(), and/or *rollback*() before the object's *onClose* event, for example, during the *canClose* event or before setting the *active* property to *false*, because *onClose* fires after the object has already closed.

Once an object has been closed, you may change its properties if desired and reopen it by setting its *active* property back to *true*.

active example

addPassword()

Related topics Example

Adds a password to the session's password list for DB table security.

Syntax

<oRef>.*addPassword*(<expC>)

<oRef>

The session you want to receive the password.

<expC>

The password string.

Property of

Session

Description

DB table security is based on password lists. If you know a password, you have access to all the files that use that password. There is no matching between a user name and password. The access level for each file may be different for the same password.

Password lists are session-based. Once a password has been added to a session, it will continue to be tried for all encrypted tables. All queries assigned to the same session in their *session* property use the same password list. If you attempt to open an encrypted table and there is no valid password that gives access to that table in the list, you will be prompted for the password, either locally in the IntraBuilder Designer or on the browser with an IntraBuilder password form. Responding with a password adds it to the list.

The *addPassword*() method allows you add passwords directly to the session's password list. You can do this if you want to add a default password, so that users won't be prompted, or if you're writing your own custom login form, and need to add the password to the session.

addPassword() example

The following *onServerClick* event handler for the login button on a custom login form adds the password typed into the password1 component and runs the main form:

```
function loginButton_onServerClick()
{
  this.form.rowset.parent.session.addPassword( this.form.password1.value );
  _sys.forms.run( "MAIN" );
}
```

append()

Related topics Example

Adds rows from one rowset or table to another.

Syntax

<oRef>.*append*()

<oRef>

The UpdateSet object that describes the append.

Property of

UpdateSet

Description

Use *append*() to add rows from a source rowset or table to an existing destination rowset or table. If there is no primary key in the destination, the rows from the source are always added. If there is a primary key in the destination, rows with keys that already exist in the destination will be copied to the table specified by the UpdateSet object's *keyViolationTableName* property instead.

To update rows with the same primary key in the destination, use the *appendUpdate*() method. To move data to a new table instead of an existing table or rowset, use the *copy*() method.

append() example

The following code accumulates records from the Daily table in an archive. The Archive table is occasionally moved to tape, so the code uses the *append*() or *copy*() method, depending on whether the Archive table already exists. The Daily table is stored in a database that supports the CURRENT_DATE SQL function.

```
d = new Database();
d.databaseName = "TRAFFIC";
d.loginString = "backup/murphy";
d.active = true;
q = new Query();
q.database = d;
q.sql = "select * from DAILY where POSTED = CURRENT_DATE";
q.active = true;
u = new UpdateSet();
u.source = q.rowset;
u.destination = "ARCHIVE.DBF";
if ( _sys.databases[ 0 ].tableExits( "ARCHIVE.DBF" ) ){
  u.append();
}
else {
  u.copy();
}
```

appendUpdate()

Related topics

Updates one rowset or table from another by updating existing rows and adding new rows.

Syntax

<oRef>.*appendUpdate*()

<oRef>

The UpdateSet object that describes the update.

Property of

UpdateSet

Description

Use *appendUpdate*() to update a rowset, allowing new rows to be added. You must specify the UpdateSet object's *indexName* property which will be used to match the records. The index must exist for the destination rowset. The original values of all changed records will be copied to the table specified by the updateSet's *changedTableName* property.

To update existing rows only, use the *update*() method instead. To always add new rows, use the *append*() method.

appendUpdate() example

applyFilter()

Related topics

Applies the filter that was set during a rowset's Filter mode.

Syntax

<oRef>.*applyFilter*()

<oRef>

The rowset whose filter criteria you want to apply.

Property of

Rowset

Description

Rowset objects support a Filter mode in which values can be assigned to Field objects and then used to filter the rows in a rowset to show only those rows with matching values. *beginFilter*() puts the rowset in Filter mode and *applyFilter*() applies the filter values. *clearFilter*() cancels the filter. Because *dataLink*ed controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a filter-by-form feature in your application.

When *applyFilter*() is called, the row cursor is repositioned to the first matching row in the set, or to the end-of-set if there are no matches. The rowset's *filter* property is updated to contain the resulting SQL expression used for the filter.

To filter rows with a condition without using Filter mode, set the rowset's *filter* property directly. See the *filter* property for more information on how filters are applied to data. To filter rows with JavaScript code instead of or in addition to an SQL expression, use the *canGetRow* event.

applyFilter() example

applyLocate()

Related topics Example

Finds the first row that matches specified criteria.

Syntax

<oRef>.*applyLocate*([<SQL condition expC>])

<oRef>

The rowset you want to search for the specified criteria.

<SQL condition expC>

An SQL condition expression.

Property of

Rowset

Description

Rowset objects support a Locate mode in which values can be assigned to Field objects and then used to find rows in a rowset that contains matching values. *beginLocate*() puts the rowset in Locate mode and *applyLocate*() finds the first matching row. *locateNext*() finds other matching rows. Because *dataLink*ed controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a search-by-form feature in your application.

applyLocate() moves the row cursor to the first row that matches the criteria set during the rowset's Locate mode.

applyLocate() also supports an optional parameter string that contains an SQL condition expression. If the parameter is used, it finds the first row that matches the condition. In either case, if no matching row is found, the row cursor is positioned at the end-of-set.

applyLocate() example

The following statement finds the first row where the City field matches the value typed into a Text component in a form. Note the use of single quotation marks to delimit the value of the Text component.

```
this.form.rowset.applyLocate( "CITY = '" + this.form.cityText.value + "'" );
```

applyUpdates()

Related topics

Attempts to apply all cached updates in the database.

Syntax

<oRef>.*applyUpdates*()

<oRef>

The database whose cached updates you want to apply.

Property of

Database

Description

applyUpdates() attempts to apply all changes to a database that have been cached and returns *true* or *false* to indicate success or failure. If it succeeds, all cached updates are cleared; if it fails, the updates remain cached. Since *applyUpdates*() uses a transaction while attempting to apply the changes and you cannot nest transactions in a database, cached updates and transaction logging with *beginTrans*() are mutually exclusive. See *cacheUpdates* for more information on caching updates.

Changes to the current row that have not been written are still in the row buffer, and have not been cached. To apply changes made to the row buffer, call the rowset's *save*() method before you call *applyUpdates*().

applyUpdates() example

Suppose you have a form that's used for redeeming prizes for points accumulated for dining at the corporate cafeteria. As each prize is chosen, the choice is written to the prize redemption table, using cached updates. The points aren't actually spent until you press the Redeem button, and you can cancel all the choices that have been made and start over by pressing the Start Over button. The following is the *onServerClick* event handler for the Redeem button.

```
function redeemButton_onServerClick()
{
  if ( this.form.rowset.save() ) {                    // Save current row
    this.form.rowset.parent.database.applyUpdates();  // Apply cached updates
  }
}
```

autoEdit

Related topics

Specifies whether the rowset defaults to Edit mode when open.

Property of

Rowset

Description

By default, a rowset's *autoEdit* property is *true*, which means the rowset will open in Edit mode. In Edit mode, data displayed in a form is immediately editable through the user interface. If you set *autoEdit* to *false*, the rowset opens in Browse mode. In Browse mode, data will be displayed in a read-only form on the browser.

To switch from Browse mode to Edit mode, call *beginEdit*(). Once the editing is complete—either saved or abandoned—the rowset goes back to Browse mode.

When *autoEdit* is *true*, the rowset can never go to Browse mode. If *autoEdit* is *true* and it is set to *false*, the rowset is switched to Browse mode.

autoEdit example

beforeGetValue

Related topics Example

Event fired when reading a field's *value* property, which returns its apparent value.

Parameters

none

Property of

CalcField, Field (including DbfField, PdxField, SqlField)

Description

By using a field's *beforeGetValue* event, you can make its *value* property appear to be anything you want. For example, in a table you can store codes, but when looking at the data, you see descriptions. The *beforeGetValue* event is also the primary way to set up a calculated field.

A field's *beforeGetValue* event handler must return a value. That value is used as the *value* property. During the *beforeGetValue* event handler, the field's *value* property represents its true value, as stored in the row buffer, which is read from the table.

Be sure to include checks for blank values—which will occur when a *beginAppend*() starts—and the end-of-set. Any attempt to access the field values when the rowset is at the end-of-set will cause an error. Return a *null* instead.

beforeGetValue is fired when reading a field's *value* property explicitly and when read to update a *dataLink*ed control. It does not fire when accessed internally for SpeedFilters, index expressions, or master-detail links, or when calling *copyToFile*().

To reverse the process, use the field's *canChange* event.

beforeGetValue example

In this example, a table of messages stores a message section number, but in the form, the section name is displayed in a Select component. To display the section name, the section number is located in the table of section numbers that is opened in the query sections1. Note the tests for the end-of-set and *beginAppend*()

```
function messages1_section_beforeGetValue()
{
  if ( this.parent.parent.endOfSet ) {
    // When navigating to end-of-set
    return null;
  }
  else if ( this.value == null ) {
    // For beginAppend()
    return "";
  }
  else {
  // Normal lookup, with value in case lookup fails
  var r = this.parent.parent.parent.parent.sections1.rowset;
  return r.applyLocate( '"Section #" = ' + parseInt( this.value ) ) ?
         r.fields[ "Name" ].value : "Closed section";
  }
}
```

beginAppend()

Related topics

Starts append of a new row.

Syntax

<oRef>.*beginAppend*()

<oRef>

The rowset you want to put in Append mode.

Property of

Rowset

Description

beginAppend() clears the row buffer and puts the rowset in Append mode, allowing the creation of a new row, either via data entry through *dataLink*ed controls or by directly assigning values to the *value* property of fields. As usual, the row buffer is not written until the rowset's *save*() method is called, there is navigation in the rowset, or the rowset's query is closed. At that point, a save attempt is made only if the rowset's *modified* property is *true*; this is intended to prevent blank rows from being added.

The integrity of the data in the row, for example making sure that all required fields are filled in, should be checked before attempting to save the row. The *abandon*() method will discard the new row, leaving no trace of the attempt.

The rowset's *canAppend* event is fired when *beginAppend*() is called. If there is a *canAppend* event handler, it must return *true* or the *beginAppend*() will not proceed. When using *beginAppend*(), you can also set up *canNavigate* and *canClose* handlers to allow or disallow navigation or closure by checking the integrity of the new row, since navigation or closure will cause the new row to be written.

The *onAppend* event is fired after the row buffer is cleared, allowing you to preset default values for any fields. After you preset values, set the *modified* property to *false*, so that the values in the fields immediately after the *onAppend* event are considered as the baseline for whether the row has been changed and needs to be saved.

An exception occurs when calling *beginAppend*() if the rowset's *live* property is *false*, or if the user has insufficient rights to add rows.

beginAppend() example

beginEdit()

Related topics

Makes contents of a row editable.

Syntax

<oRef>.*beginEdit*()

<oRef>

The rowset you want to put in Edit mode.

Property of

Rowset

Description

By default, a rowset's *autoEdit* property is *true*, which means that the rowset defaults to Edit mode when open, so *beginEdit*() is unnecessary. But you can more strictly control how editing occurs by setting *autoEdit* to *false* and calling *beginEdit*() as needed.

As usual, the row buffer is not written until the rowset's *save*() method is called, there is navigation in the rowset, or the rowset is closed. The integrity of the data in the row, for example making sure that there are no invalid entries in any fields, should be checked before attempting to save the row. The *abandon*() method will discard any changes to the row, and put the rowset back in Browse mode if the rowset's *autoEdit* property is *false*.

The rowset's *canEdit* event is fired when *beginEdit*() is called. If there is a *canEdit* event handler, it must return *true* or the *beginEdit*() will not proceed. Whether you use *beginEdit*() or leave *autoEdit* set to *true*, you can also set up *canNavigate* and *canClose* handlers to allow or disallow navigation or closure by checking the integrity of the changed row, since navigation or closure will cause the row to be written.

The *onEdit* event is fired after switching to Edit mode.

An exception occurs if the rowset's *live* property is *false*, or if the user has insufficient rights to edit rows, and they call *beginEdit*().

beginEdit() example

beginFilter()

Related topics

Puts a rowset in Filter mode, allowing the entry of filter criteria.

Syntax

<oRef>.*beginFilter*()

<oRef>

The rowset you want to put in Filter mode.

Property of

Rowset

Description

Rowset objects support a Filter mode in which values can be assigned to Field objects and then used to filter the rows in a rowset to show only those rows with matching values. *beginFilter*() puts the rowset in Filter mode and *applyFilter*() applies the filter values. *clearFilter*() cancels the filter. Because *dataLink*ed controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a filter-by-form feature in your application.

When *beginFilter*() is called, the row buffer is cleared. Values that are set either through *dataLink*ed controls or by assigning values to *value* properties are used for matching. Fields whose *value* property is left blank are not considered. To cancel Filter mode, call the *abandon*() method.

If navigation is attempted while in Filter mode, Filter mode is canceled and the navigation occurs, relative to the position of the row cursor at the time *beginFilter*() was called.

To filter rows with a condition without using Filter mode, set the rowset's *filter* property. See the *filter* property for more information on how filters are applied to data. To filter rows with JavaScript code instead of or in addition to Filter mode, use the *canGetRow* event.

beginFilter() example

beginLocate()

Related topics

Puts a rowset in Locate mode, allowing the entry of search criteria.

Syntax

<oRef>.*beginLocate*()

<oRef>

The rowset you want to put in Locate mode.

Property of

Rowset

Description

Rowset objects support a Locate mode in which values can be assigned to Field objects and then used to find rows in a rowset that contain matching values. *beginLocate*() puts the rowset in Locate mode and *applyLocate*() finds the first matching row. *locateNext*() finds other matching rows. Because *dataLink*ed controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a search-by-form feature in your application.

When *beginLocate*() is called, the row buffer is cleared. Values that are set either through *dataLink*ed controls or by assigning values to *value* properties are used for matching. Fields whose *value* property is left blank are not considered. To cancel Locate mode, call the *abandon*() method.

If navigation is attempted while in Locate mode, Locate mode is canceled and the navigation occurs, relative to the position of the row cursor at the time *beginLocate*() was called.

beginLocate() example

beginTrans()

Related topics

Begins transaction logging.

Syntax

<oRef>.*beginTrans*()

<oRef>

The database in which you want to start transaction logging.

Property of

Database

Description

Separate changes that must be applied together are considered to be a transaction. For example, transferring money from one account to another means debiting one account and crediting another. If for whatever reason one of those two changes cannot be done, the whole transaction is considered a failure and any change that was made must be undone.

Transaction logging records all the changes made to all the tables in a database. If no errors are encountered while making the individual changes in the transaction, the transaction log is cleared with the *commit*() method and the transaction is done. If an error is encountered, all changes made so far are undone by calling the *rollback*() method.

Transaction logging differs from caching updates in that changes are actually written to the disk. This means that others who are accessing the database can see your changes. In contrast, with cached updates your changes are written all at once later, when and if you decide to post the changes. For example, if you're reserving seats on an airplane, you want to post a reservation as soon as possible. If the customer changes their mind, you can undo the reservation with a rollback. With cached updates, the seat might be taken by someone else between the time the data entry for the reservation begins and the time it is actually posted.

All locks made during a transaction are maintained until the transaction is completed. This ensures that no one else can make any changes until the transaction is committed or abandoned.

For SQL-server databases, the Database object's *isolationLevel* property determines the isolation level of the transaction.

beginTrans() example

bookmark()

Related topics

Returns the current position in a rowset.

Syntax

<oRef>.*bookmark*()

<oRef>

The rowset whose current position you want to return.

Property of

Rowset

Description

A bookmark represents a position in a rowset. *bookmark*() returns the current position in the rowset. The bookmark may be stored in a variable or property so that you can go back to that position later with the *goto*() method.

A bookmark is valid only as long as the rowset stays open.

bookmark() example

cacheUpdates

Related topics

Whether to cache updates locally instead of writing to disk as they occur.

Property of

Database

Description

Normally, when a row buffer is saved, it is written to disk. By setting the *cacheUpdates* property to *true*, those changes are cached locally instead of being written to disk. One reason to do this is to reduce network traffic. Changes are accumulated and then posted with the *applyUpdates*() method, after a certain amount of time or a certain number of changes have been made.

Another reason is to simulate a transaction when you have more than one change in an all-or-nothing situation. For example, if you need to fill a customer order and reduce the stock in inventory, you cannot let one happen and not the other. When the changes are posted with *applyUpdates*(), they are applied inside a transaction at the database level. Because you cannot nest transactions, you cannot have a transaction with *beginTrans*() and use cached updates at the same time. If any of the changes do not post, for example one of the records is locked, all of the changes that did post are undone and *applyUpdates*() returns *false* to indicate failure. The cached updates remain cached so that you can retry the posting. If all the changes are posted successfully, *applyUpdates*() returns *true*.

Finally, because of the all-or-nothing nature of cached updates, you can use them to allow the user to tentatively make changes that you can simply discard as a group. For example, you could allow a user to modify a lookup table. If the user submits the changes they are applied, but if the user chooses to cancel, any changes made can be discarded by calling the *abandonUpdates*() method. Note that with cached updates, the changes aren't actually written until posted. In contrast, transaction logging actually makes the changes as they happen, but allows you to undo them if desired.

cacheUpdates example

canAbandon

Related topics

Event fired when attempt to abandon rowset occurs; return value determines if changes to row are abandoned.

Parameters

none

Property of

Rowset

Description

A rowset may be abandoned explicitly by calling its *abandon*() method, or implicitly via the user interface by pressing Esc or choosing Abandon Row from the menu while editing table rows in the IntraBuilder Designer. *canAbandon* may be used when a form is run in the IntraBuilder Designer to verify that the user wants to abandon any changes that they have made. You may check the *modified* property first to see if there are any changes to abandon; if not, there is no need to ask.

The *canAbandon* event cannot be used in the same way when a form is run from a browser, because in order to ask the user a question, they must be presented with another form (or another page of the same form), but by the time the user sees the question, the event handler would already be complete, and the row would either be abandoned or not. You could use client-side JavaScript to ask users whether they are sure they want to abandon changes, but there is no direct way to determine whether any changes have been made, as there is with the *modified* property in server-side JavaScript; so you may be asking the question unnecessarily.

Therefore *canAbandon* has a much more limited role when running a form over a browser. You could test whether the user is allowed to abandon the row, but there are few reasons why someone shouldn't be allowed to abandon a row, if they're allowed to make changes in the first place.

The *canAbandon* event handler must return *true* or *false* to indicate whether the changes to the rowset, if any, are abandoned.

canAbandon example

canAppend

Related topics

Event fired when attempting to put rowset in Append mode; return value determines if the mode switch occurs.

Parameters

none

Property of

Rowset

Description

A rowset may be put in Append mode explicitly by calling its *beginAppend*() method, or implicitly via the user interface by choosing Append Row from the menu or toolbar while editing table rows in the IntraBuilder Designer. *canAppend* may be used when a form is run in the IntraBuilder Designer to verify that the user wants to add a new row. You can check the *modified* property first to see if the user has made any changes to the current row; if not, you may not want to ask.

The *canAppend* event cannot be used in the same way when a form is run from a browser, because in order to ask the user a question, they must be presented with another form (or another page of the same form), but by the time the user sees the question, the event handler would already be complete, and the rowset would either be switched to Append mode or not. You could use client-side JavaScript to ask users whether they are sure they want to switch to Append mode, but there is no direct way to determine whether any changes have been made, as there is with the *modified* property in server-side JavaScript; so you may be asking the question unnecessarily.

Therefore *canAppend* has a more limited role when running a form over a browser. You could test whether the user is allowed to add new rows; for example, you may not allow new rows near the end of the business day.

The *canAppend* event handler must return *true* or *false* to indicate whether *beginAppend*() proceeds. *canAppend* fires before the current row is saved. If the user is editing a row, they choose to append, and *canAppend* returns *false*, the user will continue to edit that row; the buffer is untouched and is not saved. If *canAppend* returns *true*, then the *canSave* event fires. If *canSave* returns *false*, the row is not saved, and the append is canceled. If *canSave* returns *true*, then the row is saved and the append begins. This allows you to put row validation code in the *canSave* event handler that you do not need to duplicate in *canAppend*.

canAppend example

canChange

Related topics Example

Event fired when a change to the *value* property of a Field object is attempted; return value determines if the change occurs.

Parameters

<new value>

The proposed new value.

Property of

Field

Description

Use *canChange* to determine whether changes to individual fields occur. *canChange* fires when something is assigned to the *value* property of a Field object, either directly or through a *dataLink*ed control. The proposed new value is passed as a parameter to the *canChange* event handler. If the *canChange* event handler returns *false*, the Field object's *value* remains unchanged.

While *canChange* provides field-level validation to see whether changes are saved into the row buffer, use *canSave* to provide row-level validation to determine whether the buffer can be saved to disk. You should always do row-level validation no matter whether you do field-level validation or not.

You can also use *canChange* to reverse the mapping performed by *beforeGetValue*. Inside the *canChange* event handler, examine the <new value> parameter and assign the value you want to store in the table directly to the *value* property of the Field object. Doing so does not fire *canChange* recursively. Then have the *canChange* event handler return *false* so that the <new value> does not get saved into the row buffer.

canChange example

In this example, a table of messages stores a message section number, but in the form, the section name is displayed in a Select component. When a section is chosen by name, the section number is stored in the table instead with the following *canChange* event handler. The table of section numbers is opened in the query sections1.

```
function messages1_section_canChange( newValue )
{
  var r = this.parent.parent.parent.parent.sections1.rowset;  // Lookup table
  if ( r.applyLocate( '"Name" = \'' + newValue +'\'' ) ) {    // If name
found
    this.value = r.fields[ "Section #" ].value;               // save section
#
  }
  return false;    // Always return false so that newValue is not saved
}
```

canClose

Related topics

Event fired when there's an attempt to deactivate a query or stored procedure; return value determines if the object is deactivated.

Parameters

none

Property of

Query, StoredProc

Description

If the *active* property of a Query or StoredProc object is set to *false*, that object's *canClose* event fires. If the *canClose* event handler returns *false*, the close attempt fails and the *active* property remains *true*.

A StoredProc object may be deactivated only if it returns a rowset. If it returns values only, the *active* property is automatically reset to *false* after the stored procedure is called; there is nothing to deactivate.

Normally when a Query or StoredProc object closes, it saves any changes in its rowset's row buffer, if any. In attempting to save those changes, the rowset's *canSave* event is also fired, before *canClose*. If *canSave* returns *false*, the row is not saved, and the object is not closed.

If you want to abandon uncommitted changes instead of saving them when closing the object, call the rowset's *abandon*() method before closing.

canClose example

canDelete

Related topics

Event fired when attempting to delete the current row; return value determines if the row is deleted.

Parameters

none

Property of

Rowset

Description

A row may be deleted explicitly by calling the *delete*() method, or implicitly via the user interface by choosing Delete Rows from the menu or toolbar while editing table rows in the IntraBuilder Designer. *canDelete* may be used when a form is run in the IntraBuilder Designer to make sure that the user wants to delete the current row.

The *canDelete* event cannot be used in the same way when a form is run from a browser, because in order to ask the user a question, they must be presented with another form (or another page of the same form), but by the time the user sees the question, the event handler would already be complete, and the row would either be deleted or not. You could use client-side JavaScript to ask users whether they are sure they want to delete the row.

Therefore *canDelete* has a more limited role when running a form over a browser. You could test whether the user is allowed to delete the row.

The *canDelete* event handler must return *true* or *false* to indicate whether the row is deleted.

canDelete example

canEdit

Related topics

Event fired when attempting to put rowset in Edit mode; return value determines if the mode switch occurs.

Parameters

none

Property of

Rowset

Description

A rowset may be put in Edit mode explicitly by calling its *beginEdit*() method, or implicitly via the user interface by choosing Edit Row from the menu or toolbar while editing table rows in the IntraBuilder Designer. *canEdit* may be used when a form is run in the IntraBuilder Designer to verify that the user is allowed to or wants to edit the row.

The *canEdit* event handler must return *true* or *false* to indicate whether the switch to Edit mode proceeds.

canEdit example

canGetRow

Related topics Example

Event fired when attempting to read a row into the row buffer; return value determines if the row stays in or is filtered out.

Parameters

none

Property of

Rowset

Description

In addition to setting an SQL filter expression in the *filter* property, you can filter out rows through JavaScript code with *canGetRow*. In a *canGetRow* handler, the rowset acts as if the row is read into the row buffer. You can test the *value* properties of the field objects, or anything else.

If *canGetRow* returns *true*, that row is kept. If it returns *false*, the row is discarded and the next row is tried.

canGetRow example

Suppose a message database supports private messages that can be seen only by the sender and the recipient. You can prevent others from seeing private messages with a *canGetRow* event handler. The name of the user is stored as a property of the form. That name must match either the From or To fields in the message.

```
function messages1_canGetRow()
{
  return this.fields[ "From" ].value == this.parent.parent.userName ||
         this.fields[ "To"   ].value == this.parent.parent.userName;
}
```

canNavigate

Related topics

Event fired when attempting navigation in a rowset; return value determines if row cursor is moved.

Parameters

none

Property of

Rowset

Description

Navigation in a rowset may occur explicitly by calling a navigation method like *next*() or *goto*(), or implicitly via the user interface by choosing a navigation option from the menu or toolbar while viewing a rowset in the IntraBuilder Designer. *canNavigate* may be used when a form is run in the IntraBuilder Designer to verify that the user wants to leave the current row to go to another. You may check the *modified* property first to see if the user has made any changes to the current row; if not, you may not want to ask.

The *canNavigate* event handler must return *true* or *false* to indicate whether the navigation occurs. *canNavigate* fires before the current row is saved. If the user is editing a row, then they choose to navigate, and *canNavigate* returns *false*, the user will continue to edit that row; the buffer is untouched and is not saved. If *canNavigate* returns *true*, then the *canSave* event fires. If *canSave* returns *false*, the row is not saved, and the navigation is canceled. If *canSave* returns *true*, then the row is saved and the navigation occurs. This allows you to put row validation code in the *canSave* event handler that you do not need to duplicate in *canNavigate*.

canNavigate example

canOpen

Related topics

Event fired when attempting to open a query or stored procedure; return value determines if object is opened.

Parameters

none

Property of

Query, StoredProc

Description

canOpen fires when a Query or StoredProc object's *active* property is set to *true*.

If an event handler is assigned to the *canOpen* property, the event handler must return *true* or *false* to indicate whether the object is opened and activated.

canOpen example

canSave

Related topics

Event fired when attempting to save the row buffer; return value determines if the buffer is written.

Parameters

none

Property of

Rowset

Description

The row buffer may be saved explicitly by calling *save*(), or implicitly by navigating in the rowset or closing the rowset's query. Use *canSave* to verify that the data is good before attempting to write it to the disk.

The *canSave* event handler must return *true* or *false* to indicate whether the row is saved. If the user has changed the current row and attempts to append a new row or navigate, *canAppend* or *canNavigate* fires first. If that event returns *true*, then the *canSave* event fires. If *canSave* returns *false*, the row is not saved, and the attempted action does not occur. If *canSave* returns *true*, then the row is saved and the action occurs. This allows you to put row validation code in the *canSave* event handler that you do not need to duplicate in either *canAppend* or *canNavigate*.

canSave example

changedTableName

Related topics

Name of the table from which you want to collect copies of original values of rows that were changed.

Property of

UpdateSet

Description

When doing an *update*() or *appendUpdate*(), rows will be changed. The original contents of the rows that are changed are copied to the table specified by the *changedTableName* property. If the table does not exist, it is created. If it does exist, it is erased first so that it contains only those rows that were changed on the last update.

By making copies of the original values of the rows that are changed, you can undo the changes by doing another *update*(), using the *changedTableName* table as the *source* table.

changedTableName example

clearFilter()

Related topics

Clears any active filter on a rowset.

Syntax

<oRef>.*clearFilter*()

<oRef>

The rowset whose filter to clear.

Property of

Rowset

Description

clearFilter() clears the *filter* property and any filter set through the rowset's Filter mode, thereby deactivating any filters. Rows that were hidden by the filter become visible. The row cursor is not moved.

clearFilter() example

commit()

Related topics

Clears the transaction log, committing all logged changes

Syntax

<oRef>.*commit*()

<oRef>

The database whose changes you want to commit.

Property of

Database

Description

A transaction works by logging all changes. If an error occurs while attempting one of the changes, or the changes need to be undone for some other reason, the transaction is canceled by calling the *rollback*() method. Otherwise, *commit*() is called to clear the transaction log, thereby indicating that all the changes in the transaction were committed and that the transaction as a whole was posted.

commit() example

constrained

Related topics

Specifies whether updates to a rowset will be constrained by the WHERE clause of the query's SQL SELECT command. Applies to Standard tables only.

Property of

Query

Description

When *constrained* is set to *true*, any time a row is saved, if the query's SQL SELECT statement—which was stored in the *sql* property and used to generate the rowset—contains a WHERE clause, the newly saved row is evaluated against the WHERE clause. If the row no longer matches the condition set by the WHERE clause, the row is considered to be out-of-set, and the row cursor moves to the next row in the set, or to the end-of-set if already at the last row.

This property applies only to Standard tables and defaults to *false*, which means that the SQL SELECT statement is used only to generate the rowset, not to actively constrain it. By setting the *constrained* property to *true*, Standard tables behave more like SQL-server based tables, which always constrain rows according to the WHERE clause.

constrained example

copy()

Related topics

Copies one rowset or table to another rowset or table.

Syntax

<oRef>.*copy*()

<oRef>

The UpdateSet object that describes the copy.

Property of

UpdateSet

Description

The Database's *copyTable*() method is used to copy all of the rows from a single table in a database to another new table in the same database. The *copy*() method can be used for any other type of row copy: from one rowset to another in the same database, from one rowset to another in a different database, from one rowset to a table, or from one table to a rowset.

The *source* and *destination* properties specify what to copy and where to copy it. Because you can use a rowset as a *source*, you can copy only part of a table, by selecting only those rows you want to copy for the rowset. When using a table name as a *destination*, that table is created, or overwritten if it already exists. To convert from one table type to another, create a rowset of the desired result type and assign it to the *destination* property.

copy() example

copyTable()

Related topics

Makes a copy of one table to create another table in the same database.

Syntax

<oRef>.*copyTable*(<source table expC>, <destination table expC>)

<oRef>

The database in which you want to copy the table.

<source table expC>

The name of the table you want to duplicate.

<destination table expC>

The name of the table you want to create.

Property of

Database

Description

copyTable() copies all of the rows from a single table in a database to another new table in the same database. The resulting destination table will be the same table type as the source table. Use the UpdateSet's *copy*() method for any other type of row copy.

The table to copy should not be open.

To make a copy of a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].copyTable( "Stuff", "CopyOfStuff" )
```

copyTable() example

copyToFile()

Related topics

Copies the contents of a BLOB field to a new file.

Syntax

<oRef>.*copyToFile*(<file name expC>)

<oRef>

The BLOB field to copy.

<file name expC>

The name of the file you want to create.

Property of

Field

Description

copyToFile() copies the specified BLOB field to the named file.

copyToFile() example

count()

Related topics

Returns the number of rows in a rowset, respecting any filter conditions and events.

Syntax

<oRef>.*count*()

<oRef>

The rowset you want to measure.

Property of

Rowset

Description

count() returns the number of rows in the current rowset. For a rowset generated by a simple query like the following, which selects all the fields from a single table with no conditions, *count*() returns the number of rows in the table:

```
select * from CUSTOMER
```

You can use *count*() while a filter is active—with the *filter* property or the *canGetRow* event—to count the number of rows that match the filter condition. This may be time-consuming with large rowsets.

count() example

database

Related topics

The Database object to which the query or stored procedure is assigned.

Property of

Query, StoredProc

Description

A query or stored procedure must be assigned to the database that provides access to the tables it wants before it is activated. When created, a Query or StoredProc object is assigned to the default database in the default session.

To assign the object to the default database in another session, assign that session to the *session* property. Assigning the *session* property always sets the *database* property to the default database in that session.

To assign the object to another database in another session, assign the object to that session first. This makes the databases in that session available to the object.

database example

databaseName

Related topics

The BDE alias that the object represents.

Property of

Database

Description

To use a BDE alias, create a Database object and assign the alias to the object's *databaseName* property. Then set the *active* property to *true* to activate the database. While the database is active, you cannot change the *databaseName* property.

The *databaseName* property for a session's default database is always blank.

databaseName example

decimalLength

Related topics

The number of decimal places in a DBF (dBASE) numeric or float field.

Property of

DbfField

Description

The DBF (dBASE) table format supports two kinds of fields that store numbers: numeric and float. Both field types have a fixed number of decimal places. The *decimalLength* property represents the number of decimal places for any Field objects that represent a numeric or float field. For other field types, *decimalLength* is zero.

decimalLength example

delete() [Rowset]

Related topics

Deletes the current row.

Syntax

<oRef>.*delete*()

<oRef>

The rowset whose current row you want to delete.

Property of

Rowset

Description

delete() deletes the current row in the rowset. When *delete*() is called, the *canDelete* event is fired. If there is no *canDelete* event handler or the event handler returns *true*, the current row is deleted, the *onDelete* event fires, and the row cursor moves to the next row, or to the end-of-set if the last row was the one that was deleted.

While the DBF (dBASE) table format supports soft deletes, in which the rows are only marked as deleted and not actually removed until the table is packed, there is no method in the data access classes to recall those records. Therefore a *delete*() should always be considered final.

delete() [Rowset] example

delete() [UpdateSet]

Related topics

Deletes the rows in the destination that are listed in the source.

Syntax

<oRef>.*delete*()

<oRef>

The UpdateSet object that describes the delete.

Property of

UpdateSet

Description

delete() deletes the rows listed in the *source* rowset or table from the *destination* rowset or table. The *destination* must be indexed.

delete() [UpdateSet] example

destination

Related topics

The target rowset or table of an UpdateSet operation.

Property of

UpdateSet

Description

The *destination* property contains an object reference to a rowset or the name of a table that is the target of an UpdateSet operation. For an *append*(), *update*(), or *appendUpdate*(), it refers to the rowset or table that is changed. For a *copy*(), it refers to the rowset or table that receives the copies. If a table name is specified, that table is created, or overwritten if it already exists. For a *delete*(), the *destination* property refers to the table from which rows are deleted.

The *source* property specifies the other end of the UpdateSet operation.

destination example

driverName

Related topics

The database driver used for the database connection.

Property of

Database

Description

The *driverName* property reflects the database driver used for the connection. It's determined by the database driver for the database's BDE alias and set automatically once the database is successfully made active.

For default databases, the *driverName* matches the System setting in the BDE Configuration Utility.

driverName example

dropTable()

Related topics

Deletes (drops) a table from a database.

Syntax

<oRef>.*dropTable*(<table name expC>)

<oRef>

The database in which the table exists.

<table name expC>

The name of the table you want to delete.

Property of

Database

Description

dropTable() deletes a table and any existing secondary files, like memo files and indexes. *dropTable*() does not ask for confirmation; the deletion is immediate. The table cannot be open anywhere at the time of the *dropTable*(); if it is, *dropTable*() fails.

To delete a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].dropTable( "Temp" )
```

dropTable() example

emptyTable()

Related topics

Deletes all the rows in a table.

Syntax

<oRef>.*emptyTable*(<table name expC>)

<oRef>

The database in which the table exists.

<table name expC>

The name of the table you want to empty.

Property of

Database

Description

emptyTable() deletes all of the rows in a table, leaving an empty table structure, as if the table was just created. *emptyTable*() does not ask for confirmation; the deletion is immediate. The table cannot be open anywhere at the time of the *emptyTable*(); if it is, *emptyTable*() fails.

To empty a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].emptyTable( "YtdSales" )
```

emptyTable() example

endOfSet

Related topics

Specifies whether the row cursor is at the end-of-set.

Property of

Rowset

Description

The row cursor is always positioned at either a valid row or the end-of-set. There are two end-of-set positions: one before the first row and one after the last row. *endOfSet* is *true* if the row cursor is positioned at either end-of-set position.

When you first make a query active successfully, *endOfSet* is *true* if there are no rows that match the specified criteria in the query's SQL SELECT statement, or simply no rows in the tables selected.

When you apply a filter by calling *applyFilter*() or setting the *filter* property, *endOfSet* becomes *true* if there are no rows that match the filter criteria. Otherwise, the row cursor is positioned at the first matching row.

If you navigate backward before the first row in the set or after the last row in the set, this moves the row cursor to the end-of-set, so *endOfSet* becomes *true*. You can call the *first*() or *last*() methods to attempt to move the row cursor to the first or last row in the set. If after calling one of those methods, *endOfSet* is still *true*, then there are no visible rows in the current set.

Attempting to read or change a field value while at end-of-set causes an error.

endOfSet example

executeSQL()

Executes the specified SQL statement.

Syntax

<oRef>.*executeSQL*(<SQL expC>)

<oRef>

The database in which you want to execute the SQL statement.

<SQL expC>

The SQL statement.

Property of

Database

Description

Use *executeSQL*() to perform an SQL operation that does not have a data access object equivalent, for example, to use data definition language (DDL) SQL where no rowset is desired, and for server-specific SQL.

executeSQL() example

fieldName

Related topics

The name of the field represented by the Field object.

Property of

CalcField, Field (including DbfField, PdxField, SqlField)

Description

The *fieldName* property contains the name of the field that the Field object represents. The *fieldName* property is automatically filled in when the rowset object is generated.

For a CalcField object, the *fieldName* contains the name of the field, as specified when the CalcField object is created.

fieldName example

fields

Related topics

An array that contains the Field objects in a rowset.

Property of

Rowset

Description

A rowset's *fields* property contains an object reference to the array of field objects in the rowset. These fields can be accessed by their field name or their ordinal position; for example, if *this* refers to a rowset:

```
this.fields[ "State" ].value = "CA"    // Assign "CA" to State field
this.fields[ 0 ].value = 12            // Assign 12 to first field
```

To access the value of the field, you must reference the field's *value* property. You can use the *add*() method to add new CalcField objects to the *fields* array.

fields example

filter

Related topics

An SQL expression that filters out rows that do not match specified criteria.

Property of

Rowset

Description

A filter is a mechanism by which you can temporarily hide, or filter out, those rows that do not match certain criteria so that you can see only those rows that do match. The criteria is in the form of a character string that contains an SQL expression, like the one used in the WHERE clause of an SQL SELECT. For example,

```
"Firstname = 'Waldo'"
```

In this case, you would see only those rows in the current rowset whose Firstname field was “Waldo”. You can use the rowset’s Filter mode, initiated by calling the *beginFilter*() method, to build the expression automatically, and then apply it with the *applyFilter*() method. The alternative is to assign the character string directly to the *filter* property.

Setting the *filter* property causes the row cursor to move to the first matching row. If no rows match the filter expression, the row cursor is moved to the end-of-set; the *endOfSet* property is set to *true*.

While a filter is active, the row cursor will always be at either a matching row or the end-of-set. Any time you attempt to navigate to a row, the row is evaluated to see if it matches the filter condition. If it does, then the row cursor is allowed to position itself at that row and the row can be seen. If the row does not match the filter condition, the row cursor continues in the direction it was moving to find the next matching row. It will continue to move in that direction until it finds a match or gets to the end-of-set. For example, suppose that *this* is the rowset, and you add the following to your script. If no filter is active, you would move four rows forward, toward the last row:

```
this.next( 4 )
```

If a filter is active, the row cursor will move forward until it has encountered four rows that match the filter condition, and stop at the fourth. That may be the next four rows in the rowset, if they all happen to match, or the next five, or the next 400, or never, if there aren’t four rows after the current row that match. In that last case, the row cursor will be at the end-of-set.

In other words, when there is no filter active, every row is considered a match. By setting a filter, you filter out all the rows that don’t match certain criteria.

To clear a filter, you can assign an empty string to the *filter* property, or call the *clearFilter*() method.

In addition to using an SQL expression, you can filter out rows with more complex code by using the *canGetRow* event.

filter example

filterOptions

Related topics

Determines how values are matched for filtering.

Property of

Rowset

Description

The *filterOptions* property is an enumerated property that controls how the *value* properties in the field objects entered during Filter mode are matched against the values in the table. These are the options:

Value	Effect
0	Match length and case
1	Match partial length
2	Ignore case
3	Match partial length and ignore case

When matching partial length, the entire search value must match all or part of the value in the table, starting at the beginning of the field. For example, searching for “Central Park”, will match “Central Park West”, but “West” alone would not.

filterOptions also determines how fields are matched when specifying an SQL expression in the *filter* property.

filterOptions example

first()

Related topics

Moves the row cursor to the first row in the rowset.

Syntax

<oRef>.*first*()

<oRef>

The rowset in which you want to move the row cursor.

Property of

Rowset

Description

Call *first*() to move the row cursor to the first row in the rowset. If a filter is active, it moves the row cursor to the first row in the rowset that matches the filter criteria.

If the *endOfSet* property is *true* after a call to *first*(), then there are no rows that match the filter criteria if there is a filter set. If there is no filter, then that means there are no rows at all in that rowset.

first() example

goto()

Related topics

Moves the row cursor to a specific row in the rowset.

Syntax

<oRef>.*goto*(<bookmark>)

<oRef>

The rowset in which you want to move the row cursor.

<bookmark>

The bookmark you want to move to.

Property of

Rowset

Description

Call *goto*() to move the row cursor to a specific row in the rowset. You can store the current row position in a bookmark with the *bookmark*() method. You can return to that row later by calling *goto*() with that bookmark as long as the rowset has remained open. If the rowset has been closed, the bookmark is not guaranteed to return you to the correct row, since the table may have changed.

If you attempt to *goto*() a row that is out-of-set, you will generate an error.

goto() example

handle

Related topics

The BDE handle of the object.

Property of

Database, Query, Rowset, Session, StoredProc

Description

The *handle* property represents the BDE handle for the object in question. The handle can be used if you want to call BDE functions directly.

handle example

indexName [Rowset]

Related topics

The name of the index to use in the rowset.

Property of

Rowset

Description

indexName contains the name of the active controlling index tag for those table types that support index tags. It is set automatically when the query is activated to represent the tag used in the SQL SELECT's ORDER clause, if any. Assigning a new value to *indexName* supersedes any ORDER designated in the SQL SELECT statement.

The index tag is also used in a master-detail link. The index tag of the detail rowset must match the field or fields specified in the *masterFields* property.

indexName [Rowset] example

indexName [UpdateSet]

Related topics

The name of the index to use for indexed UpdateSet operations.

Property of

UpdateSet

Description

The *destination* rowset or table must be indexed for the *update*(), *appendUpdate*(), and *delete*() operations. The *indexName* property specifies the key or tag name that is to be used. For tables with primary keys, the primary key is used by default. Set the *indexName* property only if you want to use another key. For DBF (dBASE) tables, you must specify an index tag name.

indexName [UpdateSet] example

isolationLevel

Related topics

Determines the isolation level of a transaction.

Property of

Database

Description

The *isolationLevel* property is an enumerated property that determines the isolation level of a transaction. It applies to SQL-server database transactions only. For Standard table transactions, it has no effect. These are the options:

Value	Effect
0	Read uncommitted
1	Read committed
2	Repeatable read

The default is Read committed.

isolationLevel example

keyViolationTableName

Related topics

Name of the table in which you want to collect rows that could not be added because they would have caused a key violation.

Property of

UpdateSet

Description

In tables with primary keys, only one row in the table may have a particular primary key value. If the row to be added during an *append*() contains a key value that is the same as an already-existing primary key, that row cannot be added to the table, since it would have caused a primary key violation. Instead of being added to the *destination* rowset or table, that row is copied to the table specified by the *keyViolationTableName* property.

keyViolationTableName example

last()

Related topics

Moves the row cursor to the last row in the rowset.

Syntax

<oRef>.*last*()

<oRef>

The rowset in which you want to move the row cursor.

Property of

Rowset

Description

Call *last*() to move the row cursor to the last row in the rowset. If a filter is active, it moves the row cursor to the last row in the rowset that matches the filter criteria.

If the *endOfSet* property is *true* after a call to *last*(), then there are no rows that match the filter criteria if there is a filter set. If there is no filter, then that means there are no rows at all in that rowset.

Going to the last row in a rowset may not be an optimized operation on some SQL servers. For those servers, calling *last*() may take a long time for large rowsets.

last() example

length

Related topics

The maximum length of the field.

Property of

Field

Description

A field's length represents the number of bytes used in the table for that field, and for character and numeric fields, the maximum length of the item that it can store.

For character fields, the *length* property represents the maximum number of characters in the string. Attempting to store more characters in that field results in the string being truncated.

In a DBF (dBASE) table, the contents of a character field is always considered to be as long as the field itself. For example, with a field of length 10, no matter what is in the field, it is always padded with spaces at the end so that its length is 10. When working with DBF tables, you should use the *rtrim*() method to remove the trailing blanks.

For numeric fields, the *length* property represents the maximum number of characters in the number, including the digits, and any sign or decimal point. Attempting to store a number with more digits than the maximum results in numeric overflow, in which the actual value of the number is lost, and is simply considered to be bigger than the maximum allowed; it is usually represented by a string of asterisks.

length example

live

Related topics

Specifies whether the rowset can be modified.

Property of

Rowset

Description

Before making a query active, you can determine whether the rowset that is generated is editable or not. You can choose to make it not editable to prevent accidental modification of the data.

The *live* property is read-only.

live example

locateNext()

Related topics

Applies the locate criteria again to search for another row.

Syntax

<oRef>.*locateNext*([<rows expN>])

<oRef>

The rowset in which to move the row cursor.

<rows expN>

The Nth row to find. By default, the next row forward.

Property of

Rowset

Description

When the *applyLocate*() method is called, it moves the row cursor to the first row that matches the locate criteria. From then on, you can move forward and backward to other rows that match the same criteria by calling *locateNext*().

locateNext() takes an optional numeric parameter that specifies in which direction, forward or backward, to look and at which match to stop, relative to the current row position. A negative number indicates a search backward, toward the first row; a positive number indicates a search forward, toward the last row. For example, a parameter of –3 means to look backward from the current row to find the third matching row.

If the row cursor encounters the end-of-set before the desired match is found, the search stops, leaving the row cursor at the end-of-set.

locateNext() returns *true* to indicate that the desired match was found and *false* to indicate that it wasn't.

locateNext() example

locateOptions

Related topics

Determines how values are matched for locating.

Property of

Rowset

Description

The *locateOptions* property is an enumerated property that controls how the *value* properties in the field objects entered during Locate mode are matched against the values in the table. These are the options:

Value	Effect
0	Match length and case
1	Match partial length
2	Ignore case
3	Match partial length and ignore case

When matching partial length, the entire search value must match all or part of the value in the table, starting at the beginning of the field. For example, searching for “Century City”, will match “Century City East”, but “East” alone would not.

locateOptions also determines how fields are matched when using an SQL expression with the *applyLocate*() method.

locateOptions example

lockRetryCount

Related topics

The number of times to retry a lock attempt.

Property of

Session

Description

In the IntraBuilder Designer, any attempt to change the data in a row, for example, typing a letter in a *dataLink*ed text control, causes an automatic row lock to be attempted. An attempt is also made when row data changed on a client browser is posted to the IntraBuilder Server. In addition to the automatic row locking, you may request an explicit row or rowset lock with the *lockRow*() and *lockSet*() methods.

If someone else already has a conflicting lock, the initial lock attempt fails. The *lockRetryCount* property indicates the number of times the lock attempt will be retried, while the *lockRetryInterval* indicates the number of seconds to wait between each attempt. If after all the attempts the lock has not been secured, the lock request fails.

lockRetryCount example

lockRetryInterval

Related topics

The number of seconds to wait between each lock retry attempt.

Property of

Session

Description

In the IntraBuilder Designer, any attempt to change the data in a row, for example, typing a letter in a *dataLink*ed text control, causes an automatic row lock to be attempted. An attempt is also made when row data changed on a client browser is posted on the IntraBuilder Server. In addition to the automatic row locking, you may request an explicit row or rowset lock with the *lockRow*() and *lockSet*() methods.

If someone else already has a conflicting lock, the initial lock attempt fails. The *lockRetryCount* property indicates the number of times the lock attempt will be retried, while the *lockRetryInterval* indicates the number of seconds to wait between each attempt. If after all the attempts, the lock has not been secured, the lock request fails.

lockRow()

Related topics

Attempts to lock the current row.

Syntax

<oRef>.*lockRow*()

<oRef>

The rowset in which you want to lock the current row.

Property of

Rowset

Description

An automatic row lock is attempted whenever the *value* property of a Field object is modified, either directly by assignment, or indirectly through a *dataLink*ed control.

You may use *lockRow*() to attempt an explicit row lock. Whether the lock is automatic or explicit, it will fail if the current row or the entire rowset is already locked.

lockRow() returns *true* to indicate that the lock was successful and *false* to indicate that it wasn't.

Row locking support varies among different table types. The Standard (DBF and DB) tables fully support row locking; most SQL servers do not. For servers that do not support true locks, the Borland Database Engine emulates optimistic locking. Any lock request is assumed to succeed. Later, when the actual attempt to change the data occurs, if the data has changed since the lock attempt, an error occurs.

lockSet()

Related topics

Attempts to lock the entire rowset.

Syntax

<oRef>.*lockSet*()

<oRef>

The rowset you want to lock.

Property of

Rowset

Description

You may use *lockSet*() to attempt to lock the entire rowset. The rowset cannot be locked if someone else already has any other row or set locks on the rowset.

Set locks are session-based. Once a *lockSet*() attempt succeeds, all other *lockSet*( ) requests for the same set from rowsets in queries assigned to the same session will succeed. Query objects must be assigned to different Session objects for set locking to work properly.

Locking the rowset is not the same as accessing the table exclusively. Exclusive access means that you are the only one who has the table open. In contrast, locking a rowset allows others to view, but not modify, the rowset.

lockSet() returns *true* to indicate that the lock was successful and *false* to indicate that it wasn't.

Set locking support varies among different table types. The Standard (DBF and DB) tables fully support set locking, as do a few SQL servers. For servers that do not support true locks, the Borland Database Engine emulates optimistic locking. Any lock request is assumed to succeed. Later, when the actual attempt to change the data occurs, if the data has changed since the lock attempt, an error occurs.

login()

Related topics Example

Logs in user to DBF table security for a session.

Syntax

<oRef>.*login*(<group name expC>, <user name expC>, <password expC>)

<oRef>

The session to log into.

<group name expC>

The group name.

<user name expC>

The user name.

<password expC>

The password.

Property of

Session

Description

DBF table security is session-based. All queries assigned to the same session in their *session* property have the same access level. Access is assigned in one of two ways: everyone who needs to open an encrypted table will either be assigned to their own session, or they will all share the same session (for example, you might set up a guest account that everyone uses by default).

If someone attempts to open an encrypted table and has not logged in to the session, they will be prompted for the group name, user name, and password, either locally in the IntraBuilder Designer or on the browser with an IntraBuilder password form. Responding attempts to log the user into the session.

The *login*() method allows you to log in to the session directly. You can do this if you're assigning a default access level, so that users won't be prompted; or if you're writing your own custom login form, in which case you will need to call *login*() with the values you have gotten.

login() example

The following *onServerClick* event handler for the login button on a custom login form logs in the user with the values typed in the form and runs the main form:

```
function loginButton_onServerClick()
{
  this.form.rowset.parent.session.login( this.form.groupNameText.value,
                                         this.form.userNameText.value,
                                         this.form.password1.value );
  _sys.forms.run( "MAIN" );
}
```

loginString

Related topics

The user name and password to use to log in to a database.

Property of

Database

Description

Some databases require that you log in to them to access their tables. When you set the Database object's *active* property to *true* to open the connection, a login dialog will appear, prompting the user for the user name and password.

You can prevent the login dialog from appearing by setting the *loginString* property to a string containing a valid user name and password of the form "userName/password". If the user name and password provided through *loginString* are not valid, the login dialog will appear.

lookupTable

Related topics

The table used for a DB (Paradox) field's lookup.

Property of

PdxField

Description

lookupTable contains the name of the lookup table used to assist in the filling in of the field represented by the PdxField object.

lookupType

Related topics

The type of lookup used by a DB (Paradox) field.

Property of

PdxField

Description

lookupType specifies the type of lookup used to assist in the filling in of the field represented by the PdxField object.

lookupType example

masterFields

Related topics Example

A list of fields in the master rowset that link it to the detail rowset.

Property of

Rowset

Description

The *masterFields* property is set in the detail rowset. It is a string that contains a list of fields in the master rowset that are matched against the detail rowset's active controlling index, as specified by the *indexName* property. By setting the property in the detail rowset, one master rowset can control multiple detail rowsets.

The *masterRowset* property should be set before *masterFields*. Once *masterFields* is set, the detail rowset is constrained to show the detail rows that match the current row in the master rowset. You may cancel the master-detail link by setting either property to an empty string.

If there is more than one field in the list, they are separated by semicolons. You may link the rowsets through an expression by creating a calculated field in the master rowset and using that CalcField object in the *masterFields* list.

masterFields example

The following example links an employee to the various positions they have held over the years:

```
emp = new Query();
emp.sql = "select * from EMPLOYEE";
emp.active = true;
pos = new Query();
pos.sql = "select * from POSITION";
pos.active = true;
pos.rowset.indexName = "EMP_ID";
pos.rowset.masterRowset = emp.rowset;     // Identify master rowset
pos.rowset.masterFields = "EMP_ID";       // Field matches index order
```

masterRowset

Related topics Example

A reference to the master rowset that is linked the detail rowset.

Property of

Rowset

Description

The *masterRowset* property is set in the detail rowset. It is an object reference to the master rowset that constrains the detail rowset. By setting the property in the detail rowset, one master rowset can control multiple detail rowsets.

The *masterRowset* property should be set before *masterFields*. Once *masterFields* is set, the detail rowset is constrained to show the detail rows that match the current row in the master rowset. You may cancel the master-detail link by setting either property to an empty string.

masterRowset example

The following example links an employee to the various positions they have held over the years:

```
emp = new Query();
emp.sql = "select * from EMPLOYEE";
emp.active = true;
pos = new Query();
pos.sql = "select * from POSITION";
pos.active = true;
pos.rowset.indexName = "EMP_ID";
pos.rowset.masterRowset = emp.rowset;     // Identify master rowset
pos.rowset.masterFields = "EMP_ID";       // Field matches index order
```

masterSource

Related topics Example

A reference to the query that acts as master query and provides parameter values.

Property of

Query

Description

Use *masterSource* to create a master-detail link between two queries where parameters are used in the detail query.

By setting the *masterSource* property, the parameters in the SQL statement are automatically substituted with matching fields from the master query, thereby constraining the detail query. Calculated fields may be used. The fields are matched to the parameters by name. The field name match is not case-sensitive.

As navigation occurs in the *masterSource* query's rowset, the parameter values are resubstituted and the detail query is requeried.

masterSource example

Suppose you have a table of customers named CUST, and a table of their orders named ORDERS. The customers and their orders are both identified by a customer ID field, that happens (by design) to be named CUST_ID in both tables. The following statements create a master-detail link between two queries.

```
qCust = new Query();
qCust.sql = "select * from CUST";
qCust.active = true;
qOrder = new Query();
qOrder.sql = "select * from ORDERS where CUST_ID = :CUST_ID order by
ORDER_DATE";
qOrder.masterSource = qCust;
qCust.active = true;
```

The parameter CUST_ID in the SQL statement for the ORDERS table is automatically filled in with the CUST_ID field in the CUST table.

maximum

Related topics

The maximum allowed value of a DB (Paradox) field.

Property of

PdxField

Description

maximum specifies the maximum allowed value of the field represented by the PdxField object.

maximum example

minimum

Related topics

The minimum allowed value of a DB (Paradox) field.

Property of

PdxField

Description

minimum specifies the *minimum* allowed value of the field represented by the PdxField object.

minimum example

modified

Related topics Example

A flag to indicate whether the current row has been modified.

Property of

Rowset

Description

The *modified* property indicates whether the current row has been modified. It is automatically set to *true* whenever the *value* of any Field object is changed, either directly by assignment, or indirectly through a *dataLink*ed control.

If *modified* is *true*, then an attempt to save the row is made if there is navigation off the row or the rowset is closed. If *modified* is *false*, then this automatic save is not attempted.

modified is set to *false* whenever a row is read into the row buffer after navigating to it, is refreshed by *refreshRow*() or *refresh*(), or is saved. You may also set the *modified* property to *true* or *false* manually. For example, you can set *modified* to *false* after assigning some *value* properties during an *onAppend* event. This makes the values you filled in default values, and the row will not be automatically saved if the user does not add more information.

In addition to tracking changes during normal data entry, the *modified* property is also set to *true* during Filter and Locate modes. This allows you to determine if any criteria have been specified before attempting an *applyFilter*() or *applyLocate*(). When in either of these modes, navigation cancels the mode and moves the row cursor relative to the last row position, but no save is attempted, even if *modified* is *true*.

modified example

The following example is the *onServerClick* event handler for a Reply button in an E-mail viewer. It copies the name from the From field of the original to the To field of the reply and duplicates the Subject field. After setting the *value* properties, the rowset's *modified* property is set to *false* to indicate that these are the default values.

```
function replyButton_onServerClick()
{
  var cTo      = this.form.rowset.fields[ "From"    ].value;
  var cSubject = this.form.rowset.fields[ "Subject" ].value;
  this.form.rowset.beginAppend();
  this.form.rowset.fields[ "To"      ].value = cTo;
  this.form.rowset.fields[ "Subject" ].value = cSubject;
  this.form.rowset.modified = false;
}
```

next()

Related topics

Moves the row cursor to another row relative to the current position.

Syntax

<oRef>.*next*([<rows expN>])

<oRef>

The rowset in which you want to move the row cursor.

<rows expN>

The number of rows you want to move. By default, the next row forward.

Property of

Rowset

Description

next() takes an optional numeric parameter that specifies in which direction, forward or backward, to move and how many rows to move through, relative to the current row position. A negative number indicates a search backward, toward the first row; a positive number indicates a search forward, toward the last row. For example, a parameter of 2 means to move forward two rows.

If a filter is active, it is honored.

If the row cursor encounters the end-of-set while moving, the movement stops, leaving the row cursor at the end-of-set, and *next*() returns *false*. Otherwise *next*() returns *true*.

next() example

notifyControls

Related topics

Specifies whether *dataLink*ed controls are updated as field values change or the row cursor moves.

Property of

Rowset

Description

notifyControls is usually *true* so that *dataLink*ed controls are automatically updated as you navigate from row to row or when you directly assign values to the *value* property of Field objects.

You may set *notifyControls* to *false* if you are performing some data manipulation and don't want the overhead of constantly updating the controls.

When *notifyControls* is set to *true*, the controls are always refreshed, as if *refreshControls*() was called.

notifyControls example

onAbandon

Related topics

Event fired after the rowset is successfully abandoned.

Parameters

none

Property of

Rowset

Description

A rowset may be abandoned explicitly by calling its *abandon*() method, or implicitly via the user interface by pressing Esc or choosing Abandon Row from the menu while editing table rows in the IntraBuilder Designer. While the *canAbandon* event fires first to see if the abandon actually takes place, *onAbandon* fires after the abandon occurs.

If you are abandoning changes made to a row, the row is automatically refreshed, so there is no need to call *refreshRow*() in the *onAbandon*.

If you are abandoning an append initiated by *beginAppend*(), the row cursor will be at the end-of-set, so you may want to navigate to a valid row.

onAbandon example

onAppend

Related topics

Event fired after the rowset successfully enters Append mode.

Parameters

none

Property of

Rowset

Description

A rowset may be put in Append mode explicitly by calling its *beginAppend*() method, or implicitly via the user interface by choosing Append Row from the menu or toolbar while editing table rows in the IntraBuilder Designer. While the *canAppend* event fires first to see if the new append actually takes place, *onAppend* fires after the row buffer has been cleared and is ready for new values.

You can use *onAppend* to do things like automatically time stamp the new row or fill in default values. If you use *onAppend* to set field values, set the *modified* property to *false* at the end of the event handler to indicate that the row hasn't been changed by the user. This way, if the user does not add any more data, the row will not be saved automatically if they navigate to another row or try to append another.

onAppend example

onChange

Related topics

Event fired after a field's *value* property is successfully changed.

Parameters

none

Property of

Field (including DbfField, PdxField, SqlField)

Description

A Field object's *value* property may be changed directly by assigning a value to it, or indirectly through a *dataLink*ed control. When assigning a value, the change occurs during the assignment statement. When using a *dataLink*ed control, the change doesn't happen until the user tries to move the focus to another control. In both cases, *canChange* fires first to see if the change can actually take place. If it does, the value is changed and then *onChange* is fired.

onChange example

onClose

Related topics

Event fired after a query or stored procedure is successfully closed.

Parameters

none

Property of

Query, StoredProc

Description

An attempt to close a query or stored procedure occurs when its *active* property, or the *active* property of the object's database, is set to *false*. If the object's rowset has been modified, IntraBuilder will try to save it, so the close attempt can be canceled by the rowset's *canSave* event handler. If not, the row is saved.

The close can also be prevented by the Query or StoredProc object's *canClose* event handler. If not, the object is closed, and its *onClose* event fires.

Because *onClose* fires after the rowset has closed, you can no longer affect its fields. If you want to do something with the rowset's data when the rowset closes, use the *canClose* event instead, and have the event handler return *true*.

onClose example

onDelete

Related topics

Event fired after a row is successfully deleted.

Parameters

none

Property of

Rowset

Description

A row may be deleted explicitly by calling the *delete*() method, or implicitly via the user interface by choosing Delete Rows from the menu or toolbar while editing table rows in the IntraBuilder Designer. While the *canDelete* fires first to determine if the row is actually deleted, *onDelete* fires after the row has been removed.

Because the row has been removed by the time *onDelete* fires, the row cursor is at the next row or the end-of-set when *onDelete* fires.

onDelete example

onEdit

Related topics

Event fired after the rowset successfully enters Edit mode.

Parameters

none

Property of

Rowset

Description

A rowset may be put in Edit mode explicitly by calling its *beginEdit*() method, or implicitly via the user interface by choosing Edit Row from the menu or toolbar while editing table rows in the IntraBuilder Designer. While the *canEdit* event fires first to see if the switch to Edit mode actually takes place, *onEdit* fires after the rowset has switched to Edit mode.

You can use *onEdit* to do things like automatically record when edits take place, or to save original values for auditing.

onEdit example

onGotValue

Related topics

Event fired after a field's *value* property is successfully read.

Parameters

none

Property of

CalcField, Field (including DbfField, PdxField, SqlField)

Description

onGotValue is fired when reading a field's *value* property explicitly and when it is read to update a *dataLink*ed control. It does not fire when the field is accessed internally for SpeedFilters, index expressions, or master-detail links, or when calling *copyToFile*().

onGotValue example

onNavigate

Related topics

Event fired after successful navigation in a rowset.

Parameters

none

Property of

Rowset

Description

Navigation in a rowset may occur explicitly by calling a navigation method like *next*() or *goto*(), or implicitly via the user interface by choosing a navigation option from the menu or toolbar while viewing a rowset in the IntraBuilder Designer. While *canNavigate* fires first before the row cursor has moved to see if the navigation actually takes place, *onNavigate* fires after the row position has settled on the desired row or end-of-set.

Because *onNavigate* fires when moving to the end-of-set and you cannot access field values when you're at the end-of-set, you may want to test the rowset's *endOfSet* property before you attempt to access field values in your *onNavigate* handler.

You can use *onNavigate* to update non-*dataLink*ed controls or calculated fields. In that case, you may want to call your *onNavigate* handler from the *onOpen* event as well, so that these objects are up-to-date when the rowset first opens.

When navigation occurs because a row has been deleted, *onNavigate* does not fire. Call the *onNavigate* event handler from the *onDelete* event handler.

onNavigate example

onOpen

Related topics Example

Event fired after query or stored procedure is opened successfully.

Parameters

none

Property of

Query, StoredProc

Description

onOpen fires after the Query or StoredProc object has successfully opened after its *active* property has been set to *true*.

onOpen example

The following *onOpen* event handler adds a calculated field to a query.

```
function invoice1_onOpen()
{
  c = new CalcField( "Total" );
  this.rowset.fields.add( c );
  c.beforeGetValue = {||this.parent["Qty"].value *
this.parent["PricePer"].value};
}
```

Note that the *this* in the second statement refers to the query, but in the codeblock, *this* refers to the calculated field.

onSave

Related topics

Event fired after successfully saving the row buffer.

Parameters

none

Property of

Rowset

Description

The row buffer may be saved explicitly by calling *save*(), or implicitly by navigating in the rowset or closing the rowset. While *canSave* is fired first to verify that data is good before allowing it to be written, *onSave* fires after the row has been saved.

onSave example

packTable()

Related topics

Packs a Standard table by removing all deleted rows.

Syntax

<oRef>.*packTable*(<table name expC>)

<oRef>

The database in which the table exists.

<table name expC>

The name of the table you want to pack.

Property of

Database

Description

For DBF (dBASE) tables, *packTable*() removes all the records in a table that have been marked as deleted, making all the remaining records contiguous. As a result, the records are assigned new record numbers and the disk space used is reduced to reflect the actual number of records in the table.

For DB (Paradox) tables, *packTable*() removes all deleted records and redistributes the remaining records in the record blocks, optimizing the block structure.

Packing is a maintenance operation and requires exclusive access to the table; no one else may have it open at the time, or *packTable*() will fail.

To refer to a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].packTable( "Customer" )
```

packTable() example

params

Related topics Example

Parameters for an SQL statement or stored procedure call.

Property of

Query, StoredProc

Description

The *params* property contains an associative array that contains parameter names and values, if any, for an SQL statement in a Query object or a stored procedure call in a StoredProc object.

For a Query object, assigning an SQL statement with parameters to the *sql* property automatically creates the corresponding elements in the *params* array. Parameters are indicated by colons. The values you want to substitute are then assigned to the array elements in one of two ways:

- Manually, before the query is activated or requeried with *requery*().
- By assigning a *masterSource* to the query, in which case parameters are substituted with the matching fields from the *fields* array of the *masterSource*'s *rowset*. Parameters are matched to fields by name.

For a StoredProc object, the BDE will try to get the names and types of any parameters needed by a stored procedure, once the procedure name is assigned to the *procedureName* property. This works to varying degrees for most SQL servers. If it succeeds, the *params* array is filled automatically with the corresponding Parameter objects. You must then assign the values you want to substitute to the *value* property of those objects.

For SQL servers that do not return the necessary stored procedure information, include the parameters, preceded with colons, in parentheses after the procedure name. The corresponding Parameter objects in the *params* array will be created for you; then you must assign the necessary *type* and *value* information.

params example

The following statements create a query with a parameter. The parameter in the SQL statement, preceded by a colon, automatically creates the corresponding element in the *params* array.

```
q = new Query();
q.sql = "select * from CUST where CUST_ID = :custid";
q.params[ "custid" ] = 123;
q.active = true;
```

picture

Related topics

A template that formats input to a DB (Paradox) field.

Property of

PdxField

Description

A picture uses special template symbols to format data entry into a field.

picture example

precision

Related topics

The number of digits of precision of an SQL-based field.

Property of

SqlField

Description

precision represents the numeric accuracy to which numbers are stored in the field represented by the SqlField object.

precision example

prepare()

Related topics Example

Prepares an SQL statement or stored procedure.

Syntax

<oRef>.*prepare*()

<oRef>

The object you want to prepare.

Property of

Query, StoredProc

Description

prepare() prepares the stored procedure named in the *procedureName* property of a StoredProc object or the SQL statement stored in the *sql* property of a Query object. If the object is connected to an SQL-server-based database, the prepare message is passed on to the server.

Preparing an SQL statement or stored procedure call includes compiling the statement and setting up any optimizations. If the statement includes parameters, the statement can be prepared first, and, sometime later, you can get the parameter values from the client. Then the prepared statement and its parameters are ready for execution. By separating the client and server activities, things run a bit faster.

Preparing is part of the process that occurs when you set an object's *active* property to *true*, so you're never required to call *prepare*() explicitly.

prepare() example

In this example, a query is executed using a value that is entered by the user. You can prepare the query first, placing the parameter in the SQL statement with a colon in front of it:

```
q = new Query();
q.database = someDatabase;
q.sql = "select * from EMPLOYEE where EMP_ID = :id";
q.prepare();
```

Then when the user enters the value of the parameter, you assign it to the query's *params* array and execute the query:

```
q.params[ "id" ] = someForm.empIdText.value;
q.active = true;
```

Because the query has already been prepared, making it active takes less time. Later, when the parameter changes, you reassign the parameter and requery:

```
q.params[ "id" ] = someForm.empIdText.value;
q.requery();
```

problemTableName

Related topics

Name of the table in which you want to collect rows that could not be used during an update operation because of some problem other than a key violation.

Property of

UpdateSet

Description

In addition to key violations, problems during update operations are often caused by things like mismatched fields. If a row could not be transferred from the *source* to the *destination* because of a problem, it is instead copied to the table specified by the *problemTableName* property.

problemTableName example

procedureName

Related topics Example

The name of the stored procedure to call.

Property of

StoredProc

Description

Set the *procedureName* property to the name of the procedure to call. The BDE will try to get the names and types of any parameters needed by the stored procedure.

The following databases return complete parameter name and type information:

- InterBase
- Oracle
- ODBC, if the particular ODBC driver provides it

The following databases return the parameter name but not the type:

- Microsoft SQL Server
- Sybase

The following database does not return any parameter information:

- Informix

If the BDE can get the parameter names, the *params* array is filled automatically with the corresponding Parameter objects. You must then assign the values to substitute to the *value* property of those objects.

For SQL servers that do not return the necessary stored procedure information, include the parameters, preceded with colons, in parentheses after the procedure name. Empty Parameter objects will be created.

If the *type* of the parameter or the data type of the *value* for output parameters is not provided automatically, it must be set before calling the stored procedure, in addition to any input values.

procedureName example

The following statements call a stored procedure that returns an output parameter. The result is displayed in the Script Pad.

```
d = new Database();
d.databaseName = "IBLOCAL";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "DEPT_BUDGET";
p.params[ "DNO" ].value = "670";
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

The following statement calls a stored procedure in a database that does not return any parameter information. Therefore, the parameters must be declared in the *procedureName* property. Note that the parameter names are case-sensitive, and you must initialize any output parameters by assigning a dummy value of the correct data type.

```
#define PARAMETER_TYPE_INPUT        0
#define PARAMETER_TYPE_OUTPUT       1
#define PARAMETER_TYPE_INPUT_OUTPUT 2
#define PARAMETER_TYPE_RESULT       3
d = new Database();
d.databaseName = "WIDGETS";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "PROJECT_SALES( :month, :units )";
p.params[ "month" ].type  = PARAMETER_TYPE_INPUT;
p.params[ "month" ].value = 6;
p.params[ "units" ].type  = PARAMETER_TYPE_OUTPUT;
p.params[ "units" ].value = 0;                      // Output will be numeric
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

readOnly

Related topics

Whether a DBF (dBASE) or DB (Paradox) field is read-only.

Property of

DbfField, PdxField

Description

readOnly indicates whether the field represented by the Field object is read-only or not.

readOnly example

refresh()

Related topics

Refreshes data in the entire rowset.

Syntax

<oRef>.*refresh*()

<oRef>

The rowset you want to refresh.

Property of

Rowset

Description

To increase performance, rows are cached in memory as they are encountered. If the row cursor revisits a cached row, it can be reread quickly from memory instead of the disk. *refresh*() purges all cached rows —not to be confused with cached updates—for the rowset, forcing IntraBuilder to reread the data from disk. It discards any changes to the row buffer, so a row that has been modified is not saved. When the rowset is refreshed, any *dataLink*ed controls are also refreshed with values for the current row if *notifyControls* is *true*.

refresh() does not regenerate the rowset. If the rowset is not *live*, *refresh*() has no effect. Use *requery*() to regenerate the rowset.

refresh() example

refreshControls()

Related topics

Refreshes any controls that are *dataLink*ed to the current row.

Syntax

<oRef>.*refreshControls*()

<oRef>

The rowset you want to refresh.

Property of

Rowset

Description

refreshControls() updates any controls that are *dataLink*ed to Field objects in the rowset, regardless of the setting of the *notifyControls* property. The controls are updated with the values in the row buffer, not the values on disk.

Use *refreshRow*() first to refresh the fields in the row buffer with the values on disk if desired.

refreshControls() example

refreshRow()

Related topics

Refreshes data in the current row.

Syntax

<oRef>.*refresh*()

<oRef>

The rowset you want to refresh.

Property of

Rowset

Description

refreshRow() rereads the data for the current row from disk. It discards any changes to the row buffer, so a row that has been modified is not saved. When the row is refreshed, any *dataLink*ed controls are also refreshed if *notifyControls* is *true*.

Use *refresh*() to refresh the entire rowset.

refreshRow() example

reindex()

Related topics

Rebuilds a Standard table's indexes from scratch.

Syntax

<oRef>.*reindex*(<table name expC>)

<oRef>

The database in which the table exists.

<table name expC>

The name of the table you want to reindex.

Property of

Database

Description

Indexes can become unbalanced during normal use. Occasionally, they can also be corrupted. In both cases, you can fix the problem by rebuilding the indexes, as if you were creating them from scratch.

Reindexing is a maintenance operation and requires exclusive access to the table; no one else may have it open at the time, or *reindex*() will fail.

To refer to a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].reindex( "Customer" )
```

reindex() example

renameTable()

Related topics

Renames a table in a database.

Syntax

<oRef>.*renameTable*(<original name expC>, <new name expC>)

<oRef>

The database in which to rename the table.

<original name expC>

The original name of the table.

<new name expC>

The new name of the table.

Property of

Database

Description

renameTable() renames a table in a database, including all secondary files such as index and memo files.

The table to rename should not be open.

To rename a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].renameTable( "Before", "After" )
```

renameTable() example

replaceFromFile()

Related topics

Copies the contents of a file into a BLOB field.

Syntax

<oRef>.*replaceFromFile*(<file name expC> [,<append expL>])

<oRef>

The BLOB field you want to copy into.

<file name expC>

The name of the file you want to copy.

<append expL>

Whether to append the new data or overwrite.

Property of

Field

Description

replaceFromFile() copies the contents of the named field into the specified BLOB field.

By specifying <append expL> as *true*, the contents of the file are added to the end of the current contents of the BLOB field. If the parameter is specified as *false* or left out, the BLOB field will be overwritten and end up containing only the contents of the file.

replaceFromFile() example

requery()

Related topics Example

Re-executes the query or stored procedure, regenerating the rowset.

Syntax

<oRef>.*requery*()

<oRef>

The query or stored procedure you want to re-execute.

Property of

Query, StoredProc

Description

requery() re-executes a stored procedure or a query's SQL statement, generating an up-to-date rowset. Calling *requery*() is similar to setting the object's *active* property to *false* and back to *true*, except that *requery*() does not prepare the SQL statement. This includes attempting to save the current row if necessary and closing the object, firing all the events along the way. If those actions are halted by the *canSave* or *canClose* event handlers, the *requery*() attempt will stop at that point.

Use *requery*() when a parameter in the SQL statement has changed to re-execute the query with the new value.

Use *refresh*() to refresh the rowset without re-executing the query, which is faster. But *refresh*() has no effect on a rowset that is not *live*; use *requery*() instead.

requery() example

In this example, a query is executed using a value that is entered by the user. You can prepare the query first, placing the parameter in the SQL statement with a colon in front of it:

```
q = new Query();
q.database = someDatabase;
q.sql = "select * from EMPLOYEE where EMP_ID = :id";
q.prepare();
```

Then when the user enters the value of the parameter, you assign it to the query's *params* array and execute the query:

```
q.params[ "id" ] = someForm.empIdText.value;
q.active = true;
```

Because the query has already been prepared, making it active takes less time. Later, when the parameter changes, you reassign the parameter and requery:

```
q.params[ "id" ] = someForm.empIdText.value;
q.requery();
```

requestLive

Related topics

Specifies whether the query should generate an editable rowset.

Property of

Query

Description

Before making a query active, you can determine whether the rowset that is generated is editable or not. You can choose to make it not editable to prevent accidental modification of the data.

requestLive defaults to *true*.

requestLive example

required

Related topics

Whether a DB (Paradox) field is required to be filled in and not left blank.

Property of

PdxField

Description

required indicates whether the field represented by the Field object is a required field; that is, whether it must be filled in.

required example

rollback()

Related topics

Cancels the transaction by undoing all logged changes

Syntax

<oRef>.*rollback*()

<oRef>

The database whose changes you want to rollback.

Property of

Database

Description

A transaction works by logging all changes. If an error occurs while attempting one of the changes, or the changes need to be undone for some other reason, the transaction is canceled by calling the *rollback*() method. Otherwise, *commit*() is called to clear the transaction log, thereby indicating that all the changes in the transaction were committed and that the transaction as a whole was posted.

Since new rows have already been written to disk, rows that were added during the transaction are deleted. In the case of DBF (dBASE) tables, the rows are marked as deleted, but are not physically removed from the table. If you want to actually remove them, you can pack the table with *packTable*(). Rows that were just edited are returned to their saved values.

All locks made during a transaction are maintained until the transaction is completed. This ensures that no one else can make any changes until the transaction is committed or abandoned.

rollback() example

rowset

Related topics

A reference to the query's or stored procedure's rowset.

Property of

Query, StoredProc

Description

A Query object always contains a *rowset* property, but that property does not refer to a valid Rowset object until the query has been activated and the rowset has been opened.

Some stored procedures generate rowsets. If that is the case, the StoredProc object's *rowset* property refers to that rowset after the stored procedure is executed.

The *rowset* property is read-only.

rowset example

save()

Related topics

Saves the current row buffer.

Syntax

<oRef>.*save*()

<oRef>

The rowset you want to save.

Property of

Rowset

Description

A row is saved automatically if it has been modified and there is either navigation in the rowset or the rowset is closed. You may call *save*() explicitly to write the row buffer. By design, *save*() has no effect if the rowset's *modified* property is *false*, because supposedly there are no changes to save; and a successful *save*() sets the *modified* property to *false*, to reflect that the values in the controls are the ones on disk. You can manipulate the *modified* property to control this designed behavior.

The *canSave* event fires after calling *save*(). If there is no *canSave* event handler, or *canSave* returns *true*, then the row buffer is saved, the *modified* property is set to *false*, and the *onSave* event fires.

The row cursor does not move after a *save*() unless the values that were saved cause the row to become out-of-set, in which case the row cursor is moved to the next available row, or the end-of-set if there are no more available rows.

Changes are written to disk unless the *cacheUpdates* property is set to *true*, in which case the changes are cached. Whether the changes are actually written to a physical disk depends on the operating system and its own disk caches, if any.

save() example

scale

Related topics

The scale of an SQL-based field.

Property of

SqlField

Description

scale represents the scale of the field represented by the SqlField object.

scale example

session

Related topics

The Session object to which the database, query, or stored procedure is assigned.

Property of

Database, Query, StoredProc

Description

A database must be assigned to a session. When created, a Database object is assigned to the default session.

A query or stored procedure must be assigned to a database, which in turn is assigned to a session. When created, a Query or StoredProc object is assigned to the default database in the default session.

To assign the object to the default database in another session, assign that session to the *session* property. Assigning the *session* property always sets the *database* property to the default database in that session.

To assign the object to another database in another session, assign the object to that session first. This makes the databases in that session available to the object.

session example

source

Related topics

The source rowset or table of an UpdateSet operation.

Property of

UpdateSet

Description

The *source* property contains an object reference to a rowset or the name of a table that is the source of an UpdateSet operation. For an *append*(), *update*(), or *appendUpdate*(), it refers to the rowset or table that contains the new data. For a *copy*(), it refers to the rowset or table that is to be duplicated. For a *delete*(), the *source* property refers to the table that contains the list of rows to be deleted.

The *destination* property specifies the other end of the UpdateSet operation.

source example

sql

Related topics Example

The SQL statement that describes the query.

Property of

Query

Description

The *sql* property of a Query object contains an SQL SELECT statement that describes the rowset to be generated. To use a stored procedure in an SQL server that returns a rowset, use the *procedureName* property of a StoredProc object instead.

The *sql* property must be assigned before the Query object is activated.

The SQL SELECT statement may contain an ORDER BY clause to set the row order, a WHERE clause to select a subset of rows, perform a JOIN, or any other SQL SELECT clause.

But to take full advantage of the data access objects' features—such as locating and filtering—with SQL-server-based tables, the SQL SELECT used to access a table must be a simple SELECT: all the fields from a single table, with no options. For example,

```
select * from CUSTOMER
```

If the SQL statement is not a simple SELECT, locating and filtering is performed locally, instead of by the SQL server. If the result of the SELECT is a small rowset, local searching will be fast; but if the result is a large rowset, local searching will be slow. For large rowsets, you should use a simple SELECT, or use parameters in the SQL statement and *requery*() as needed instead of relying on the Locate and Filter features.

Master-detail linking through the *masterRowset* and *masterFields* properties with SQL-server-based tables also requires a simple SELECT. An alternative is master-detail linking though Query objects with the *masterSource* property and parameters in the SQL statement. There is no simple SELECT restriction when using Standard tables.

Parameters in an SQL statement are indicated by a colon. For example,

```
select * from CUST where CUST_ID = :cust_id
```

Whenever the SQL property is assigned, it is scanned for parameters. IntraBuilder automatically creates corresponding elements in the query's *params* array, with the name of the parameter as the array index. For more information, see the *params* property.

In addition to assigning the SQL statement directly to the *sql* property, you may also use an SQL statement in an external file. To use an external file, place an "@" symbol before the file name in the *sql* property. For example,

```
@ORDERS.SQL
```

The external file must be either a query file created by the Visual Query Builder with a QRY extension, or a text file that contains an SQL statement. For a text file, an SQL extension is assumed.

sql example

The following SQL statement will select all the fields in the table MESSAGES:

```
select * from MESSAGES
```

state

Related topics Example

An enumerated value indicating the rowset's current mode.

Property of

Rowset

Description

The *state* property is read-only, indicating which mode the rowset is in, as listed in the following table:

Value	Mode
0	Closed
1	Browse
2	Edit
3	Append
4	Filter
5	Locate

- When the rowset's query is not active, the rowset is Closed.
- While the query is active, if the rowset's *autoEdit* property is *false*, the rowset is in Browse mode, if it's not in one of the next four modes.
- If the rowset's *autoEdit* property is *true*, the rowset is in Edit mode while the query is active and not in one of the next three modes; it's never in Browse mode. If *autoEdit* is *false*, then the rowset is in Edit mode only after a successful *beginEdit*() and it stays in that mode until the row is saved or abandoned.
- After a successful *beginAppend*(), it is in Append mode. It stays in that mode until the new row is saved or abandoned.
- After a *beginFilter*(), it is in Filter mode. It stays in that mode until there is an *applyFilter*() or the Filter mode is abandoned.
- After a *beginLocate*(), it is in Locate mode. It stays in that mode until there is an *applyLocate*() or the Locate mode is abandoned.

state example

The following *onServerClick* event handler for a button labeled “Apply” tests the rowset’s *state* property so that it calls either *applyFilter*() or *applyLocate*(), depending on the rowset’s current mode. It uses manifest constants created with the #define preprocessor directive (and available in the INTRA.H include file) to represent the options of the *state* property, which makes the code more readable.

```
#define STATE_CLOSED   0
#define STATE_BROWSE   1
#define STATE_EDIT     2
#define STATE_APPEND   3
#define STATE_FILTER   4
#define STATE_LOCATE   5
Ä
function applyButton_onServerClick()  // Apply Filter or Locate
{
  switch( this.form.rowset.state ) {
    case STATE_FILTER:
      this.form.rowset.applyLocate();
      break;
    case STATE_LOCATE:
      this.form.rowset.applyFilter();
      break;
  }
}
```

tableExists()

Related topics

Checks to see if a specified table exists in a database.

Syntax

<oRef>.*tableExists*(<table name expC>)

<oRef>

The database in which to see if the table exists.

<table name expC>

The name of the table you want to look for.

Property of

Database

Description

tableExists() returns *true* if a table with the specified name exists in the database.

To look for a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the _sys object. For example,

```
_sys.databases[ 0 ].tableExists( "Billing" )
If you do not specify an extension, IntraBuilder will look for both a DBF
(dBASE) and DB (Paradox) table with that name.
```

tableExists() example

type [Field]

Related topics

The data type of the value stored in a field.

Property of

CalcField, Field (including DbfField, PdxField, SqlField)

Description

The *type* property reflects the data type stored in the field represented by the Field object.

type [Field] example

type [Parameter]

Related topics

An enumerated value indicating the type of parameter.

Property of

Parameter

Description

The *type* property indicates the type of parameter a Parameter object represents, as listed in the following table:

Value	Type
0	Input
1	Output
2	InputOutput
3	Result

See the Parameter object's *value* property for details on each type.

state example

unidirectional

Related topics

Specifies whether to assume forward-only navigation to increase performance on SQL-based servers.

Property of

Query

Description

If *unidirectional* is set to *true*, previously visited rows are not cached and less communication is required between IntraBuilder and the SQL server. This results in fewer resources consumed and better performance, but is worthwhile only if you never want to go backward in the rowset.

If *unidirectional* is *true*, you may still be able to go backward, depending on the server, but if so it would be time-consuming.

unidirectional example

unlock()

Related topics

Releases row and rowset locks.

Syntax

<oRef>.*unlock*()

<oRef>

The rowset that contains the lock.

Property of

Rowset

Description

unlock() releases automatic row locks and locks set by *lockRow*() and *lockSet*()

You cannot release locks during a transaction.

unlock() example

update()

Related topics

Updates existing rows in one rowset from another.

Syntax

<oRef>.*update*()

<oRef>

The UpdateSet object that describes the update.

Property of

UpdateSet

Description

Use *update*() to update a rowset. You must specify the UpdateSet object's *indexName* property that will be used to match the records. The index must exist for the destination rowset. The original values of all changed records will be copied to the table specified by the UpdateSet object's *changedTableName* property.

To add new rows and update existing rows only, use the *appendUpdate*() method instead.

update() example

updateWhere

Related topics

Determines which fields to use in constructing the WHERE clause in an SQL UPDATE statement. SQL-based servers only.

Property of

Query

Description

The *updateWhere* property may be one of the following values: AllFields, KeyFields, or KeyFieldsAndChangedFields.

updateWhere example

user()

Related topics

Returns the login name of the user currently logged in to the session.

Syntax

<oRef>.*user*()

<oRef>

The session you want to check.

Property of

Session

Description

user() returns the login name of the user currently logged in to a session on a system that has DBF table security in place. If no DBF table security has been configured, or no one has logged in to the session, *user*() returns an empty string.

user() example

value [CalcField]

Related topics

The value of a calculated field.

Property of

CalcField

Description

The *value* property of a CalcField object reflects its current value.

To set the value of a CalcField object, you can do one of two things:

- Assign a code-reference, either a codeblock or function pointer, to the CalcField object's *beforeGetValue* event. The return value of the code becomes the CalcField object's value.
- Assign a value to the CalcField object's *value* property directly as needed, like in the rowset's *onNavigate* event.

value [CalcField] example

The first example uses the CalcField's *beforeGetValue* event to calculate the total price from the quantity and price per item for each line item:

```
q = new Query();
q.sql = "select * from LINEITEM";
q.active = true;
c = new CalcField( "Total" );
q.rowset.fields.add( c );
c.beforeGetValue = {||this.parent["Quantity"].value *
this.parent["PricePer"].value};
```

Because *this* refers to the CalcField object itself, *this*.*parent* refers to the *fields* array, through which you can access the other field objects. The second example adds a CalcField to an existing *fields* array:

```
q.rowset.fields.add( new CalcField( "Commission" ) );
```

It then uses the following code in the rowset's *onNavigate* event to set the CalcField object's *value* property:

```
this.fields[ "Commission" ].value = this.fields[ "SellPrice" ].value / 10;
```

value [Field]

Related topics Example

The value of a field in the row buffer.

Property of

Field (including DbfField, PdxField, SqlField)

Description

All of the Field objects in the rowset's *fields* array property have a *value* property, which reflects the value of the field in the row buffer, which in turn reflects the values of the fields in the current row.

You may attempt to change the value of a *value* property directly by assignment, in which case the attempt occurs immediately, or through a *dataLink*ed control, in which case the attempt occurs when the control loses focus. In either case, the field's *canChange* property fires to see whether the change is allowed. If *canChange* returns *false*, then the assignment doesn't take; if the change was through a *dataLink*ed control, the control still contains the proposed new value. If *canChange* returns *true* or there is no *canChange* event handler, the field's value is changed and the *onChange* event fires.

When a field is changed, the rowset's *modified* property is automatically set to *true* to indicate that the rowset has been changed.

By using a field's *beforeGetValue* event, you can make the *value* property appear to be something else besides what is in the row buffer.

value [Field] example

The following example is the *onServerClick* event handler for a Reply button in an E-mail viewer. It copies the name from the From field of the original to the To field of the reply and duplicates the Subject field. After setting the *value* properties, the rowset's *modified* property is set to *false* to indicate that these are the default values.

```
function replyButton_onServerClick()
{
  var cTo      = this.form.rowset.fields[ "From"    ].value;
  var cSubject = this.form.rowset.fields[ "Subject" ].value;
  this.form.rowset.beginAppend();
  this.form.rowset.fields[ "To"      ].value = cTo;
  this.form.rowset.fields[ "Subject" ].value = cSubject;
  this.form.rowset.modified = false;
}
```

value [Parameter]

Related topics Example

The input, output, or result value of a stored procedure.

Property of

Parameter

Description

Values are transmitted to and from stored procedures through Parameter objects. Each object's *type* property indicates what type of parameter the object represents. Depending on which one of the four types the parameter is, its *value* property is handled differently.

- Input: an input value for the stored procedure. The *value* must be set before the stored procedure is called.
- Output: an output value from the stored procedure. The *value* must be set to the correct data type before the stored procedure is called; any dummy value may be used. Calling the stored procedure sets the *value* property to the output value.
- InputOutput: both input and output. The *value* must be set before the stored procedure is called. Calling the stored procedure updates the *value* property with the output value.
- Result: the result value of the stored procedure. In this case, the stored procedure acts like a function, returning a single result value, instead of updating parameters that are passed to it. Otherwise, the *value* is treated like an output value. The name of the Result parameter is always "Result".

If a Parameter object is assigned as the *dataLink* of a component in a form, changes to the component are reflected in the *value* property of the Parameter object, and updates to the *value* property of the Parameter object are displayed in the component.

value [Parameter] example

The following statements call a stored procedure that returns an output parameter. The result is displayed in the Script Pad.

```
d = new Database();
d.databaseName = "IBLOCAL";
d.active = true;
p = new StoredProc();
p.database = d;
p.procedureName = "DEPT_BUDGET";
p.params[ "DNO" ].value = "670";                    // Set input parameter
p.active = true;
_sys.scriptOut.writeln( p.params[ "TOT" ].value );  // Display output
```

Form objects

IntraBuilder forms perform a wide range of tasks, from publishing Web pages to functioning as interactive database applications that host Java applets and ActiveX controls. The Form Expert and Form Designer allow you to create and modify forms visually. Forms are saved as JavaScript code in a JFM file that you can modify.

All forms start with a Form object, which acts as a container for all the other objects in the form. Depending on what the form does, it will contain a combination of the following groups of objects:

- Data access objects, which give access to data in tables and allow data publishing and data entry
- Query objects
- Database objects
- Session objects
- Visual components for data display and entry
- Button objects
- CheckBox objects
- Hidden objects
- HTML objects
- Image objects
- ListBox objects
- Password objects
- Radio objects
- Reset objects
- Rule objects
- Select objects
- Text objects
- TextArea objects

  Use the component that's appropriate for the data; for example, CheckBox objects for *true*/*false* values, Select objects to pick one item from a list, and Text objects for basic data entry. IntraBuilder supports all HTML form components for data display and entry.

- Web components, that extend the capabilities of your application
- Java applets
- ActiveX controls

  Identify and configure these components to run when the form is viewed on a browser.

Link visual components to data access objects through the components' *dataLink* property for automatic data reading and writing. Tie everything else together with IntraBuilder's enhanced version of JavaScript.

class ActiveX

Related topics

A rectangular region on a form set aside to contain an ActiveX control when the form is run on a client browser.

Syntax

[<oRef> =] new ActiveX(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created ActiveX object.

<form>

The Form object to which you're binding the ActiveX object.

Properties

The following tables list the properties, events, and methods of the ActiveX class.

Property	Default	Description
alt		An alternate string that is displayed if the client browser does not support ActiveX controls
classId		The ID string that identifies the ActiveX control
className	ActiveX	Identifies the object as an instance of the ActiveX class
codeBase		The URL for the ActiveX control
form		The form that contains the ActiveX object
height		Height in characters
left		The location of the left edge of the ActiveX object in characters, relative to the left edge of the form
name		The name of the ActiveX object
pageno	1	The page of the form on which the ActiveX object appears
params		Parameters passed to the ActiveX control
parent		The form that contains the ActiveX object
top		The location of the top edge of the ActiveX object in characters, relative to the top edge of the form
width		Width in characters

Event	Parameters	Description
onDesignLoad	<from palette expL>	After the ActiveX object is first added from the palette and then every time the form is opened in the Form Designer
onServerLoad		After the form containing the ActiveX object is loaded, but before it is rendered into HTML

Method	Parameters	Description
release()		Explicitly releases the ActiveX object from memory

Description

An ActiveX object in IntraBuilder is a place holder for an ActiveX control, not an actual ActiveX control.

To include an ActiveX control in a form when it is rendered in HTML, create an ActiveX object on the form. Set the *codeBase* property to the URL of the ActiveX control and the *classId* property to the component's ID string. Declare parameters if needed in the *params* property.

If the ActiveX control needs client-side setup, you can use the form's *onLoad* event.

class ActiveX example

class Button

Related topics

A button on a form.

Syntax

[<oRef> =] new Button(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Button object.

<form>

The Form object to which you're binding the Button object.

Properties

The following tables list the properties, events, and methods of the Button class.

Property	Default	Description
className	Button	Identifies the object as an instance of the Button class
form		The form that contains the button
left	0	The location of the left edge of the button in characters, relative to the left edge of the form
name		The name of the button
pageno	1	The page of the form on which the button appears
parent		The form that contains the button
text	<same as name>	The text that appears on the button face
top	0	The location of the top edge of the button in characters, relative to the top edge of the form
visible	true	Whether the button is visible
width		Width in characters

Event	Parameters	Description
onClick		Client-side: after the button is clicked
onDesignLoad	<from palette expL>	After the button is first added from the palette and then every time the form is opened in the Form Designer
onServerClick		After the button is clicked
onServerLoad		After the form containing the button is loaded, but before it is rendered into HTML

Method	Parameters	Description
release()		Explicitly releases the Button object from memory

Description

Use a Button object to execute an action when the user clicks it. A Button object has both an *onClick* event and an *onServerClick* event. The *onClick* event is intended to fire on the browser, but it will fire when the form is used in the IntraBuilder Designer.

Most Button objects have an *onServerClick* event handler only. When the button is clicked on the browser, the form is submitted and the button's *onServerClick* event handler fires. In HTML, the text that appears in the button to determine which button was clicked to submit the form, so you cannot have two or more buttons with the same *text* property appear on a form at the same time. Those Buttons objects may exist, but only one can be visible at any time; the rest must be invisible or on another page of the form.

If a Button object has both an *onClick* and *onServerClick* event handler, the *onClick* event will occur on the browser. The form is not submitted—unless the *onClick* event handler calls the form's *submit*() method—so the *onServerClick* event will not occur when the button is clicked on the browser.

class Button example

class CheckBox

Related topics

A check box on a form. CheckBox objects may also appear on reports.

Syntax

[<oRef> =] new CheckBox(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created CheckBox object.

<form>

The Form object to which you're binding the CheckBox object.

Properties

The following tables list the properties, events, and methods of the CheckBox class.

Property	Default	Description
checked	false	Whether the check box is visually marked
className	CheckBox	Identifies the object as an instance of the CheckBox class
color	black	The color of the checkbox's text label
dataLink		The Field object that is linked to the CheckBox
fontBold	true	Whether the check box's text label appears in bold face
fontItalic	false	Whether the check box's text label appears italicized
fontName	MS San Serif	The typeface of the check box's text label
fontStrikeout	false	Whether the check box's text label appears struck through
fontUnderline	false	Whether the check box's text label is displayed underlined
form		The form that contains the check box
height		Height in characters
left		The location of the left edge of the check box in characters, relative to the left edge of the form
name		The name of the check box
pageno	1	The page of the form on which the check box appears
parent		The form that contains the check box
text	<same as name>	The text label that appears beside the check box
top		The location of the top edge of the check box in characters, relative to the top edge of the form
visible	true	Whether the check box is visible
width		Width in characters

Event	Parameters	Description
canRender		Reports only: before the checkbox is rendered; return value determines whether checkbox is rendered
onClick		Client side: after the checkbox is clicked
onDesignLoad	<from palette expL>	After the checkbox is first added from the palette and then every time the form is opened in the Form Designer
onRender		Reports only: after the checkbox is rendered
onServerLoad		After the form containing the checkbox is loaded, but before it is rendered into HTML

Method	Parameters	Description

release()	Explicitly releases the CheckBox object from memory

Description

Use a CheckBox component to represent a *true*/*false* value.

class CheckBox example

class Form

A Form object.

Syntax

[<oRef> =] new Form([<title expC>])

<oRef>

A variable or property in which to store a reference to the newly created Form object.

<title expC>

An optional title for the Form object. If not specified, the title will be “Form”.

Properties

The following tables list the properties, events, and methods of the Form class.

Property	Default	Description
background		Background image
bodyTag		Extra attributes to include in the <BODY> tag of the form
className	Form	Identifies the object as an instance of the Form class
color	silver	Background color
elements		An array containing object references to the components on the form
gridLineWidth	0	Width of HTML table grid lines when form is displayed on the browser (0=no grid lines)
headTag		Extra tags to include in the <HEAD> section of the document
height		Height in characters
left		The location of the left edge in characters
linkColor		The color of hyperlinks
pageno	1	The current page in the form
rowset		The primary rowset
title	Form	The title of the report; appears in the title bar
top		The location of the top edge in characters
virtualRoot		Base directory from which files are accessed by the form
vlinkColor		The color of visited hyperlinks
width		Width in characters

Event	Parameters	Description
onDesignLoad		After the form is first opened in the Form Designer
onLoad		Client side: after the entire document has been loaded
onServerLoad		After the form has been opened
onServerSubmit		After the form has been submitted by the client-side *submit*() method
onServerUnload		After the form has been closed
preRender		Before the form is rendered

Method	Parameters	Description
close()		Closes and unloads the form
open()		Loads and opens the form
release()		Explicitly releases the Form object from memory
submit()		Submits the form

Description

A Form object acts as a container for other visual components and the data access objects that are linked to them.

An object reference to all the visual components in a form is stored in its *elements* array. All of the visual components have a *form* property that points back to the form.

The form has a *rowset* property that refers to its primary rowset. Components can access this rowset in their event handlers generically with the object reference *this.form.rowset*. For example, a button on a form that goes to the first row in the rowset would have an *onServerClick* event handler like this:

```
function firstButton_onServerClick()
{
  this.form.rowset.first();
}
```

If the form has more than one rowset, each one can be addressed through the *rowset* property of the Query objects, which are properties of the form. For example, to go to the last row in the rowset of the Query object members1, the *onServerClick* event handler would look like this:

```
function lastMemberButton_onServerClick()
{
  this.form.members1.rowset.last();
}
```

class Form example

class Hidden

Related topics

A component on a form that has no visual representation, but maintains a value.

Syntax

[<oRef> =] new Hidden(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Hidden object.

<form>

The Form object to which you're binding the Hidden object.

Properties

The following tables list the properties, events, and methods of the Hidden class.

Property	Default	Description
className	Hidden	Identifies the object as an instance of the Hidden class
form		The form that contains the Hidden object
left		The location of the left edge of the Hidden object in characters, relative to the left edge of the form
name		The name of the Hidden object
pageno	1	The page of the form on which the Hidden object appears
parent		The form that contains the Hidden object
top		The location of the top edge of the Hidden object in characters, relative to the top edge of the form
value		The value maintained by the Hidden object

Method	Parameters	Description
release()		Explicitly releases the Hidden object from memory

Event	Parameters	Description
onDesignLoad	<from palette expL>	After the Hidden object is first added from the palette and then every time the form is opened in the Form Designer
onServerLoad		Fires on the server after the form containing the Hidden object is loaded, but before it is rendered into HTML

Description

A Hidden object has no visual representation when the form is run, although it can be seen in the Form Designer. It is intended to maintain a hidden value that can be modified by the browser with client-side JavaScript.

class Hidden example

class HTML

Related topics

Non-editable HTML text on a form. HTML objects may also appear on reports.

Syntax

[<oRef> =] new HTML(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created HTML object.

<form>

The Form object to which you're binding the HTML object.

Properties

The following tables list the properties, events, and methods of the HTML class.

Property	Default	Description
alignHorizontal	Left	Determines how the text displays within the horizontal plane of its rectangular frame (0=Left, 1=Center, 2=Right, 3=Justify)
alignVertical	Top	Determines how the text displays in the vertical plane of its rectangular frame (0=Top, 1=Center, 2=Bottom, 3=Justify)
borderStyle	Default	Specifies whether a box border appears (0=Normal, 1=Raised, 2=Lowered, 3=None, 4=Single, 5=Double, 6-Drop Shadow, 7=Client, 8=Modal, 9=Etched In, 10=Etched Out). (See page 14-25.)
className	HTML	Identifies the object as an instance of the HTML class
color	black	The color of the text
fixed	false	Whether the HTML object's position is fixed or if it can be "pushed down" or "pulled up" by the rendering or suppression of other objects
fontBold	true	Whether the HTML object's text appears in bold face
fontItalic	false	Whether the HTML object's text appears italicized
fontName	MS San Serif	The typeface of the HTML object's text
fontStrikeout	false	Whether the HTML object's text appears struck through
fontUnderline	false	Whether the HTML object's text is displayed underlined
form		The form that contains the HTML object
height		Height in characters
leading	0	The distance between consecutive lines; if 0 uses the font's default leading
left	0	The location of the left edge of the HTML object in characters, relative to the left edge of the form
marginHorizontal		The horizontal margin between the text and its rectangular frame
marginVertical		The vertical margin between the text and its rectangular frame
name		The name of the HTML object
pageno	1	The page of the form on which the HTML object appears
parent		The form in which the HTML object is contained
rotate	0	The text orientation, in increments of 90 degrees (0=0, 1=90, 2=180, 3=270)
suppressIfBlank	false	Whether the HTML object is suppressed (not rendered) if it is blank
suppressIfDuplicate	false	Whether the HTML object is suppressed (not rendered) if its value is the same as the previous time it was rendered
template		Formatting template
text	<same as name>	The value of the HTML object; the text that appears
top	0	The location of the top edge of the HTML object in characters, relative

		to the top edge of the form
tracking	0	The space between characters; if 0 uses the font's default
trackJustifyThreshold	0	The maximum amount of added space between words on a fully justified line; 0 indicates no limit
variableHeight	false	Whether the HTML object's height can increase based on its value
verticalJustifyLimit	0	The maximum additional space between lines that can be added to attempt to justify vertically. If the limit is exceeded the HTML object is top justified. A value of 0 means no limit.
visible	true	Whether the HTML object is visible
width	10	Width in characters

Event	Parameters	Description
canRender		Reports only: before the HTML object is rendered; return value determines whether the HTML object is rendered
onDesignLoad	<from palette expL>	After the HTML object is first added from the palette and then every time the form is opened in the Form Designer
onRender		Reports only: after the HTML object is rendered
onServerLoad		After the form containing the HTML object is loaded, but before it is rendered into HTML

Method	Parameters	Description
release()		Explicitly releases the HTML object from memory

Description

Use an HTML component to display information in a form or report. The *text* property of the component may contain any text, including HTML tags.

The *text* property may be an expression codeblock, which is evaluated every time it is rendered.

class HTML example

class Image

Related topics

A rectangular region on a form that displays a bitmap image. Image objects may also appear on reports.

Syntax

[<oRef> =] new Image(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Image object.

<form>

The Form object to which you're binding the Image object.

Properties

The following tables list the properties, events, and methods of the Image class.

Property	Default	Description
alignment	Stretch	Determines the size and position of the graphic inside the Image object (0=Stretch, 1=Top left, 2=Centered, 3=Keep aspect stretch, 4=True size)
className	Image	Identifies the object as an instance of the Image class
dataSource		The file or field that is displayed in the Image object
form		The form that contains the Image object
height		Height in characters
left		The location of the left end of the Image object in characters, relative to the left edge of the form
name		The name of the Image object
pageno	1	The page of the form on which the Image object appears
parent		The form that contains the Image object
top	0	The location of the top edge of the Image object in characters, relative to the top edge of the form
visible	true	Whether the Image object is visible
width		Width in characters

Event	Parameters	Description
canRender		Reports only: before the Image object is rendered; return value determines whether Image object is rendered
onDesignLoad	<from palette expL>	After the Image object is first added from the palette and then every time the form is opened in the Form Designer
onImageClick		Client-side: after the Image object is clicked
onImageServerClick	<left expN>, <top expN>	After the image object is clicked.
onRender		Reports only: after the Image object is rendered
onServerLoad		After the form containing the Image object is loaded, but before it is rendered into HTML

Method	Parameters	Description
move()		Repositions and/or resizes the Image object
release()		Explicitly releases the Image object from memory

Description

Use an Image object to display a bitmap image. The image can be data from a field, or a static image like a company logo.

You can also create controls with images. Clicking on the image will fire either the object's *onImageClick* event on the browser, or the *onImageServerClick* event on the server. The *onImageServerClick* event gets the coordinates of the pixel that was clicked, so that you can create image maps.

class Image example

class JavaApplet

Related topics

A rectangular region on a form set aside to contain a Java applet when the form is run on a client browser.

Syntax

[<oRef> =] new JavaApplet(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created JavaApplet object.

<form>

The Form object to which you're binding the JavaApplet object.

Properties

The following tables list the properties, events, and methods of the JavaApplet class.

Property	Default	Description
alt		An alternate string that is displayed if the client browser does not support Java applets
className	JavaApplet	Identifies the object as an instance of the JavaApplet class
code		The access function of the Java applet
codeBase		The URL for the Java applet
form		The form that contains the JavaApplet object
height		Height in characters
left	0	The location of the left edge of the JavaApplet object in characters, relative to the left edge of the form
name		The name of the JavaApplet object
pageno	1	The page of the form on which the JavaApplet object appears
params		Parameters passed to the Java applet
parent		The form that contains the JavaApplet object
top		The location of the top edge of the JavaApplet object in characters, relative to the top edge of the form
width		Width in characters

Event	Parameters	Description
onDesignLoad	<from palette expL>	After the JavaApplet object is first added from the palette and then every time the form is opened in the Form Designer
onServerLoad		After the form containing the JavaApplet object is loaded, but before it is rendered into HTML

Method	Parameters	Description
release()		Explicitly releases the JavaApplet object from memory

Description

A JavaApplet object in IntraBuilder is a place holder for a Java applet, not an actual Java applet.

To include a Java applet in a form when it is rendered in HTML, create a JavaApplet object on the form. Set the *codeBase* property to the URL of the Java applet and the *code* property to the name of the access function. Declare parameters if needed in the *params* property.

class JavaApplet example

class ListBox

Related topics

A selection list on a form, from which you can pick multiple items.

Syntax

[<oRef> =] new ListBox(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created ListBox object.

<form>

The Form object to which you're binding the ListBox object.

Properties

The following tables list the properties, events, and methods of the ListBox class.

Property	Default	Description
className	ListBox	Identifies the object as an instance of the ListBox class
form		The form that contains the ListBox object
height		Height in characters
left		The location of the left edge of the ListBox object in characters, relative to the left edge of the form
multiple	false	Whether the ListBox object allows selection of more than one item
name		The name of the ListBox object
options		The options strings of the ListBox object
pageno	1	The page of the form on which the select appears
parent		The form that contains the ListBox object
selected		An array of the option(s) marked as selected
top	0	The location of the top edge of the select in characters, relative to the top edge of the form
value		The value of the option that currently has focus
visible	true	Whether the ListBox object is visible
width		Width in characters

Event	Parameters	Description
onBlur		Client-side: after the ListBox object loses focus
onChange		Client-side: after the selection has changed and the ListBox object loses focus, but before onBlur
onDesignLoad	<from palette expL>	After the select is first added from the palette and then every time the form is opened in the Form Designer
onFocus		After the ListBox object gains focus
onServerLoad		Fires on the server after the form containing the select is loaded, but before it is rendered into HTML

Method	Parameters	Description
focus()		Client-side: sets focus to the ListBox object
release()		Explicitly releases the ListBox object from memory

Description

Use a ListBox object to present the user with a scrollable list of items. If the *multiple* property is *true*, the user can choose more than one item. The list of options is set with the *options* property. The list of items

selected is returned in the *selected* array.

class ListBox example

class Password

Related topics

A single-line text input field on a form that obscures the text that is typed into it.

Syntax

[<oRef> =] new Password(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Password object.

<form>

The Form object to which you're binding the Password object.

Properties

The following tables list the properties, events, and methods of the Password class.

Property	Default	Description
className	Password	Identifies the object as an instance of the Password class
dataLink		The Field object that is linked to the Password object
form		The form that contains the Password object
left		The location of the left edge of the Password object in characters, relative to the left edge of the form
name		The name of the Password object
pageno	1	The page of the form on which the Password object appears
parent		The form that contains the Password object
top		The location of the top edge of the Password object in characters, relative to the top edge of the form
value		The obscured string contained in the Password object
visible	true	Whether the Password object is visible
width		Width in characters

Event	Parameters	Description
onDesignLoad	<from palette expL>	After the Password object is first added from the palette and then every time the form is opened in the Form Designer
onServerLoad		After the form containing the Password object is loaded, but before it is rendered into HTML

Method	Parameters	Description
focus()		Client-side: sets focus to the Password object
release()		Explicitly releases the Password object from memory

Description

The Password component is similar to the Text component, except that it obscures the characters that are typed into it. Use it for passwords and other sensitive information that should not be seen by an observer.

class Password example

class Radio

Related topics

A single radio button on a form. The user may choose one from a set. Radio objects may also appear on reports.

Syntax

[<oRef> =] new Radio(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Radio object.

<form>

The Form object to which you're binding the Radio object.

Properties

The following tables list the properties, events, and methods of the Radio class.

Property	Default	Description
className	Radio	Identifies the object as an instance of the Radio class
color	black	The color of the radio button's text label
dataLink		The Field object that is linked to the Radio object
fontBold	true	Whether the radio button's text label appears in bold face
fontItalic	false	Whether the radio button's text label appears italicized
fontName	MS San Serif	The typeface of the radio button's text label
fontStrikeout	false	Whether the radio button's text label appears struck through
fontUnderline	false	Whether the radio button's text label is displayed underlined
form		The form that contains the radio button
groupName		The group to which the radio button belongs
height		Height in characters
left		The location of the left edge of the radio button in characters, relative to the left edge of the form
name		The name of the radio button
pageno	1	The page of the form on which the radio button appears
parent		The form that contains the radio button
text	<same as name>	The text label that appears beside the radio button
top		The location of the top edge of the radio button in characters, relative to the top edge of the form
value	true	Whether the radio button is visually marked as selected
visible	true	Whether the radio button is visible
width		Width in characters

Event	Parameters	Description
canRender		Reports only: before the radio button is rendered; return value determines whether radio button is rendered
onDesignLoad	<from palette expL>	After the radio button is first added from the palette and then every time the form is opened in the Form Designer
onRender		Reports only: after the radio button is rendered
onServerLoad		After the form containing the radio button is loaded,

but before it is rendered into HTML

Method	Parameters	Description
focus()		Client-side: sets focus to the radio button
release())		Explicitly releases the radio button from memory

Description

Use a group of Radio objects to present the user a set of multiple choices, from which they can choose only one.

Each set of choices on a form must have the same *groupName* property. If there is only one group of radio buttons on a form, the *groupName* can be left blank.

class Radio example

class Reset

Related topics

A button on a form that resets the form values to their defaults.

Syntax

[<oRef> =] new Reset(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Reset object.

<form>

The Form object to which you're binding the Reset object.

Properties

The following tables list the properties, events, and methods of the Reset class.

Property	Default	Description
className	Reset	Identifies the object as an instance of the Reset class
form		The form that contains the Reset object
left		The location of the left edge of the Reset object in characters, relative to the left edge of the form
name		The name of the Reset object
pageno	1	The page of the form on which the Reset object appears
parent		The form that contains the Reset object
text	<same as name>	The text that appears on the Reset object's face
top		The location of the top edge of the Reset object in characters, relative to the top edge of the form
visible	true	Whether the Reset object is visible
width		Width in characters

Event	Parameters	Description
onClick		Client-side: after the Reset object is clicked on the client
onDesignLoad	<from palette expL>	After the Reset object is first added from the palette and then every time the form is opened in the Form Designer
onServerLoad		After the form containing the Reset object is loaded, but before it is rendered into HTML

Method	Parameters	Description
release()		Explicitly releases the Reset object from memory

Description

A Reset button has only one function: to reset the components on a form to their original values. This occurs in the browser; the form is not submitted.

class Reset example

class Rule

Related topics

A horizontal line, or rule, on a form. Rule objects may also appear on reports.

Syntax

[<oRef> =] new Rule(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Rule object.

<form>

The Form object to which you're binding the Rule object.

Properties

The following tables list the properties, events, and methods of the Rule class.

Property	Default	Description
className	Rule	Identifies the object as an instance of the Rule class
form		The form that contains the rule
left		The location of the left end of the rule in characters, relative to the left edge of the form
name		The name of the rule
pageno	1	The page of the form on which the rule appears
parent		The form in which the rule is contained
right		The location of the right end of the rule in characters, relative to the left edge of the form
size	1	Width in pixels
top		The vertical location of the rule in characters, relative to the top edge of the form

Event	Parameters	Description
canRender		Reports only: before the rule is rendered; return value determines whether the rule is rendered
onDesignLoad	<from palette expL>	After the rule is first added from the palette and then every time the form is opened in the Form Designer
onRender		Reports only: after the rule is rendered
onServerLoad		After the form containing the rule is loaded, but before it is rendered into HTML

Method	Parameters	Description
release()		Explicitly releases the Rule object from memory

Description

Use a Rule object to draw a horizontal line in a form or report.

class Rule example

class Select

Related topics

A component on a form which can be temporarily expanded to show a list from which you can pick a single item.

Syntax

[<oRef> =] new Select(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Select object.

<form>

The Form object to which you're binding the Select object.

Properties

The following tables list the properties, events, and methods of the Select class.

Property	Default	Description
className	Select	Identifies the object as an instance of the Select class
dataLink		The Field object that is linked to the Select object
form		The form that contains the Select object
left		The location of the left edge of the Select object in characters, relative to the left edge of the form
name		The name of the Select object
options		The options strings of the Select object
pageno	1	The page of the form on which the Select object appears
parent		The form that contains the Select object
top		The location of the top edge of the Select object in characters, relative to the top edge of the form
value		The value of the currently selected option
visible	true	Whether the Select object is visible
width		Width in characters

Event	Parameters	Description
onBlur		Client-side: after the Select object loses focus
onChange		Client-side: after the selection has changed and the Select object loses focus, but before *onBlur*
onDesignLoad	<from palette expL>	After the Select object is first added from the palette and then every time the form is opened in the Form Designer
onFocus		Client-side: after the Select object gains focus
onServerLoad		After the form containing the select is loaded, but before it is rendered into HTML

Method	Parameters	Description
focus()		Client-side: sets focus to the Select object
release())		Explicitly releases the Select object from memory

Description

Use a Select object when you want the user to pick one item from a list. When the user is not choosing an item, the list is not visible. The list of options is set with the *options* property.

class Select example

class Text

Related topics

A single-line text input field on a form.

Syntax

[<oRef> =] new Text(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Text object.

<form>

The Form object to which you're binding the Text object.

Properties

The following tables list the properties, events, and methods of the Text class.

Property	Default	Description
className	Text	Identifies the object as an instance of the Text class
dataLink		The Field object that is linked to the Text object
form		The format that contains the Text object
left		The location of the left edge of the Text object in characters, relative to the left edge of the form
name		The name of the Text object
pageno	1	The page of the form on which the Text object appears
parent		The form that contains the Text object
template		Formatting template
top		The location of the top edge of the Text object in characters, relative to the top edge of the form
value		The string currently displayed in the Text object
visible	true	Whether the Text object is visible
width		Width in characters

Event	Parameters	Description
onBlur		Client-side: after the Text object loses focus
onChange		Client-side: after the string in the Text object has changed and the Text object loses focus, but before *onBlur*
onDesignLoad	<from palette expL>	After the Text object is first added from the palette and then every time the form is opened in the Form Designer
onFocus		Client-side: after the Text object gains focus
onSelect		Client-side: after a selection in the Text object is started
onServerLoad		After the form containing the Text object is loaded, but before it is rendered into HTML

Method	Parameters	Description
focus()		Client-side: sets focus to the Text object
release()		Explicitly releases the Text object from memory

Description

Text objects are the primary data display and entry component. They display the contents of a field in a single line entry field that can be edited.

class Text example

class TextArea

Related topics

A multiple-line text input field on a form.

Syntax

[<oRef> =] new TextArea(<form>)

<oRef>

A variable or property—typically of <form>—in which to store a reference to the newly created Text object.

<form>

The Form object to which you're binding the TextArea object.

Properties

The following tables list the properties, events, and methods of the TextArea class.

Property	Default	Description
className	TextArea	Identifies the object as an instance of the TextArea class
dataLink		The Field object that is linked to the TextArea object
form		The form that contains the TextArea object
height		Height in characters
left		The location of the left edge of the TextArea object in characters, relative to the left edge of the form
name		The name of the TextArea object
pageno	1	The page of the form on which the TextArea object appears
parent		The form that contains the TextArea object
readOnly	false	Whether changes in the TextArea object will be saved
top		The location of the top edge of the TextArea object in characters, relative to the top edge of the form
value		The string currently displayed in the TextArea object
visible	true	Whether the TextArea object is visible
width		Width in characters

Event	Parameters	Description
onBlur		Client-side: after the TextArea object loses focus
onChange		Client-side: after the string in the TextArea object has changed and the TextArea object loses focus, but before onBlur
onDesignLoad	<from palette expL>	After the TextArea object is first added from the palette and then every time the form is opened in the Form Designer
onFocus		Client-side: after the TextArea object gains focus
onSelect		Client-side: after a selection in the TextArea object is started
onServerLoad		Fires on the server after the form containing the TextArea object is loaded, but before it is rendered into HTML

Method	Parameters	Description
focus()		Client-side: sets focus to the TextArea object
release()		Explicitly releases the TextArea object from memory

Description

Use a TextArea component to display and edit multi-line text. To display the text but not allow changes, set the *readOnly* property to *true*. The user will be able to make changes in their browser, but the changes will not be saved.

class TextArea example

alignHorizontal

Related topics

Determines the horizontal alignment of text in an HTML component.

Property of

HTML

Description

alignHorizontal determines the way the text displays within the horizontal plane of its rectangular frame. Set it to one of the following:

Value	Alignment
0	Left
1	Center
2	Right
3	Justify

alignment

Related topics

Determines the size and position of the graphic inside an Image object.

Property of

Image

Description

If a graphic is smaller than the Image object that displays it, it can be stretched to fill the Image object or positioned inside the Image object with empty space around it. Assign one of the following settings to the *alignment* property of an Image object to determine how the graphic is aligned.

Setting	Description
0 (Stretch)	Enlarge graphic to fill the entire Image object
1 (Top Left)	In the top left corner of the Image object
2 (Center)	Centered in the Image object
3 (Keep Aspect Stretch)	Maintains the original height/width (aspect) ratio when stretching the graphic until it fills either dimension of the Image object
4 (True Size)	No changes to the graphic

If the graphic is larger than the Image object, both Stretch and Keep Aspect Stretch will reduce the graphic to fit the Image object so that the entire image is visible. Top Left and Center will both display whatever fits in the Image object.

True Size does not change the graphic at all. The Image object is dynamically resized to display the graphic. This is the fastest option, because IntraBuilder doesn't have to do any stretching or shrinking.

alignVertical

Related topics

Determines the vertical alignment of text in an HTML component.

Property of

HTML

Description

alignVertical determines the way the text displays within the vertical plane of its rectangular frame. Set it to one of the following:

Value	Alignment
0	Top
1	Middle
2	Bottom
3	Justify

The Justify option has no effect when the object is rendered as HTML. The object is top-aligned instead.

alt

Example

The string that is displayed when a client browser does not support Java applets.

Property of

JavaApplet

Description

If a browser does not support Java applets, or if Java is disabled, an HTML string can be displayed in place of the Java applet.

Set the *alt* property to an appropriate message. For example, you can remind the user to make sure Java support is enabled, suggest they upgrade their browser, or point them to a version of your form that does not have a Java applet.

alt example

Suppose you're using a Java applet to display a graph. The following string contains a link to a version of the form that does not rely on a Java applet and displays the numbers in a table instead.

```
The graph requires Java support. <A HREF="TABLE.JFM">Just show me the
numbers.</A>
```

background

Related topics

A form's background image.

Property of

Form

Description

Set the *background* property to the file name of a bitmap you want tiled in the background of your form.

You may use any IntraBuilder-supported bitmap format. IntraBuilder will convert the file to GIF or JPG on-the-fly if necessary.

Setting a background image supersedes the background color designated by the form's *color* property.

bodyTag

Related topics Example

Extra attributes to include in the <BODY> tag of the form.

Property of

Form

Description

Use the *bodyTag* property to include extra attributes in the <BODY> tag of the HTML document generated by the IntraBuilder Agent. The contents of the *bodyTag* property are included as is, just before the closing angle bracket of the <BODY> tag.

The extra attributes would not include any angle brackets, since they are attributes inside an HTML tag.

You can also include extra HTML after the <BODY> tag by starting the *bodyTag* property with the closing angle bracket—that ends the <BODY> tag—and then whatever HTML you want. A closing bracket—the one that was supposed to end the <BODY> tag—will always be added after the *bodyTag*. Be careful not to disrupt the structure of the document generated by IntraBuilder.

bodyTag example

On some browsers, you can specify that the background image position remains fixed and does not scroll with the rest of the document by setting the BGPROPERTIES attribute to “fixed” in the <BODY> tag. This attribute is not directly supported by IntraBuilder, but you can include it by setting the form’s *bodyTag* property to

```
bgproperties="fixed"
```

The resulting <BODY> tag looks like this:

```
<body bgproperties="fixed">
```

checked

Related topics Example

Specifies whether a CheckBox object is visually marked.

Property of

CheckBox

Description

The *checked* property contains *true* if the CheckBox object is checked or *false* if it is not. You can read the *checked* property to determine the CheckBox object's current state, or assign true or false to simulate checking and unchecking the CheckBox.

If the CheckBox object is *dataLink*ed to a field, reading and writing the *checked* property has the same effect as reading and writing to the field.

checked example

Suppose you have a number of check boxes for options. You want an All button that checks all the options and a None button that unchecks them all. You can create a client-side function that both buttons call in their *onClick* handlers to set the check boxes.

```
function allButton_onClick()
{
  checkAll( true );
}
function noneButton_onClick()
{
  checkAll( false );
}
function checkAll( lArg );
{
// {Export} This comment causes this function body to be sent to the client
  // Set all checkboxes to the designated value
  var f = document.forms[ 0 ];  // Get reference to form
  f.peanutsCheck.checked   = lArg;
  f.sprinklesCheck.checked = lArg;
  f.fudgeCheck.checked     = lArg;
  // etc.
}
```

classId

Related topics Example

The ID string of an ActiveX control.

Property of

ActiveX

Description

To use an ActiveX control in a form, set the *codeBase* property to the relative or absolute URL for the control and the *classId* property to the control's ID string. The string should start with the letters "clsid:"

Parameters for the ActiveX control can be set in the *params* property.

classId example

The following is a code excerpt from a JFM form file that creates an ActiveX component that defines the control's *classId*, *codeBase*, and a number of parameters in the *params* array:

```
with (this.activex1 = new ActiveX(this)) {
   height = 10;
   left = 4;
   top = 1;
   width = 30;
   classId = "clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2";
   codeBase = "http://activex.microsoft.com/controls/iexplorer/ielabel.ocx";
   params["Angle"]="45";
   params["Alignment"]="4";
   params["FontBold"]="1";
   params["ForeColor"]="#ffa500";
   params["FontName"]="Arial";
   params["BackStyle"]="0";
   params["Caption"]="Welcome!";
   params["FontSize"]="20";
}
```

close()

Related topics Example

Closes a form.

Syntax

<oRef>.*close*()

<oRef>

An object reference to the form to close.

Property of

Form

Description

Use *close*() to close an open form.

When running on the IntraBuilder Agent, closing a form removes it from the form stack. If it was the form on the top of the form stack, the form underneath it on the form stack is displayed on the browser. If it was the only form on the form stack, an error is displayed on the browser, since there are no more forms to display.

close() example

The following is an *onServerClick* event handler for a button that launches another form and closes the current one.

```
function viewerButton_onServerClick()
{
  _sys.forms.run( "VIEWER" );  // Launch the Viewer form
  this.form.close();           // Close the current form
}
```

code

Related topics Example

The access function of a Java applet.

Property of

JavaApplet

Description

To use a Java applet in a form, set its *codeBase* property to the relative or absolute URL for the component and the *code* property to the name of the access function.

Parameters for the Java applet can be set in the *params* property.

code example

The following is a code excerpt from a JFM form file that creates a JavaApplet component.

```
with (this.javaapplet1 = new JavaApplet(this)) {
   code = "TicTacToe.class";
   codeBase = "http://java.sun.com/applets/applets/TicTacToe";
}
```

codeBase

Related topics Example

The URL (Uniform Resource Locator) for a Java applet or ActiveX control.

Property of

ActiveX, JavaApplet

Description

To use a Java applet or ActiveX control in a form, set its *codeBase* property to the relative or absolute URL for the component.

In addition, for a Java applet set the *code* property to the name of the access function.

Parameters for the Java applet or ActiveX control can be set in the *params* property.

codeBase example

The following is a code excerpt from a JFM form file that creates a JavaApplet component.

```
with (this.javaapplet1 = new JavaApplet(this)) {
   code = "TicTacToe.class";
   codeBase = "http://java.sun.com/applets/applets/TicTacToe";
}
```

color

Related topics Example

The color of an object.

Property of

CheckBox, Form, HTML, Radio

Description

The *color* property contains the color of an object:

- For a Form object, the *color* is the form's background color.
- For a CheckBox, HTML, or Radio object, the *color* is the color of the text.

The value of *color* may be any hexadecimal RGB (Red Green Blue) triplet or a JavaScript color name. Neither of them are case-sensitive.

JavaScript color names and RGB values

Color	Red	Green	Blue
aliceblue	F0	F8	FF
antiquewhite	FA	EB	D7
aqua	00	FF	FF
aquamarine	7F	FF	D4
azure	F0	FF	FF
beige	F5	F5	DC
bisque	FF	E4	C4
black	00	00	00
blanchedalmond	FF	EB	CD
blue	00	00	FF
blueviolet	8A	2B	E2
brown	A5	2A	2A
burlywood	DE	B8	87
cadetblue	5F	9E	A0
chartreuse	7F	FF	00
chocolate	D2	69	1E
coral	FF	7F	50
cornflowerblue	64	95	ED
cornsilk	FF	F8	DC
crimson	DC	14	3C
cyan	00	FF	FF
darkblue	00	00	8B
darkcyan	00	8B	8B
darkgoldenrod	B8	86	0B
darkgray	A9	A9	A9
darkgreen	00	64	00
darkkhaki	BD	B7	6B
darkmagenta	8B	00	8B
darkolivegreen	55	6B	2F
darkorange	FF	8C	00
darkorchid	99	32	CC
darkred	8B	00	00
darksalmon	E9	96	7A

darkseagreen	8F	BC	8F
darkslateblue	48	3D	8B
darkslategray	2F	4F	4F
darkturquoise	00	CE	D1
darkviolet	94	00	D3
deeppink	FF	14	93
deepskyblue	00	BF	FF
dimgray	69	69	69
dodgerblue	1E	90	FF
firebrick	B2	22	22
floralwhite	FF	FA	F0
forestgreen	22	8B	22
fuchsia	FF	00	FF
gainsboro	DC	DC	DC
ghostwhite	F8	F8	FF
gold	FF	D7	00
goldenrod	DA	A5	20
gray	80	80	80
green	00	80	00
greenyellow	AD	FF	2F
honeydew	F0	FF	F0
hotpink	FF	69	B4
indianred	CD	5C	5C
indigo	4B	00	82
ivory	FF	FF	F0
khaki	F0	E6	8C
lavender	E6	E6	FA
lavenderblush	FF	F0	F5
lawngreen	7C	FC	00
lemonchiffon	FF	FA	CD
lightblue	AD	D8	E6
lightcoral	F0	80	80
lightcyan	E0	FF	FF
lightgoldenrodyellow	FA	FA	D2
lightgreen	90	EE	90
lightgrey	D3	D3	D3
lightpink	FF	B6	C1
lightsalmon	FF	A0	7A
lightseagreen	20	B2	AA
lightskyblue	87	CE	FA
lightslategray	77	88	99
lightsteelblue	B0	C4	DE
lightyellow	FF	FF	E0
lime	00	FF	00
limegreen	32	CD	32
linen	FA	F0	E6

magenta	FF	00	FF
maroon	80	00	00
mediumaquamarine	66	CD	AA
mediumblue	00	00	CD
mediumorchid	BA	55	D3
mediumpurple	93	70	DB
mediumseagreen	3C	B3	71
mediumslateblue	7B	68	EE
mediumspringgreen	00	FA	9A
mediumturquoise	48	D1	CC
mediumvioletred	C7	15	85
midnightblue	19	19	70
mintcream	F5	FF	FA
mistyrose	FF	E4	E1
moccasin	FF	E4	B5
navajowhite	FF	DE	AD
navy	00	00	80
oldlace	FD	F5	E6
olive	80	80	00
olivedrab	6B	8E	23
orange	FF	A5	00
orangered	FF	45	00
orchid	DA	70	D6
palegoldenrod	EE	E8	AA
palegreen	98	FB	98
paleturquoise	AF	EE	EE
palevioletred	DB	70	93
papayawhip	FF	EF	D5
peachpuff	FF	DA	B9
peru	CD	85	3F
pink	FF	C0	CB
plum	DD	A0	DD
powderblue	B0	E0	E6
purple	80	00	80
red	FF	00	00
rosybrown	BC	8F	8F
royalblue	41	69	E1
saddlebrown	8B	45	13
salmon	FA	80	72
sandybrown	F4	A4	60
seagreen	2E	8B	57
seashell	FF	F5	EE
sienna	A0	52	2D
silver	C0	C0	C0
skyblue	87	CE	EB
slateblue	6A	5A	CD

slategray	70	80	90
snow	FF	FA	FA
springgreen	00	FF	7F
steelblue	46	82	B4
tan	D2	B4	8C
teal	00	80	80
thistle	D8	BF	D8
tomato	FF	63	47
turquoise	40	E0	D0
violet	EE	82	EE
wheat	F5	DE	B3
white	FF	FF	FF
whitesmoke	F5	F5	F5
yellow	FF	FF	00
yellowgreen	9A	CD	32

color example

Both of the following strings represent the color orange and can be used as the *color* property:

```
0xffa500
orange
```

dataLink

Related topics

The Field object that is linked to the component.

Property of

CheckBox, Radio, Password, Select, Text, TextArea

Description

You link a form component to a table's field by assigning a reference to the *dataLink* property of the component. The reference you assign is to the Field object that represents the field in an open query. This assignment is called *dataLink*ing. When a form component and Field object are linked in this way, they are said to be *dataLink*ed.

Both field and component objects have a *value* property. (For a CheckBox object, its value is its *checked* property, but the concept is the same.) When they are *dataLink*ed, changes in one object's *value* property are echoed in the other. The form component's *value* property reflects the value displayed in the component at any given moment. If the component's value is changed, it is copied into the field, either after the component loses focus (if you're running the form locally in the IntraBuilder Designer) or when the entire form is submitted (if you're running the form remotely on a browser).

The *value* property for all fields in a rowset are set when you first open a query and updated as you navigate from row to row. The *value* properties for components *dataLink*ed to those fields are also updated at the same time, unless the rowset's *notifyControls* property is set to *false*. You can also force the components to be updated by calling the rowset's *refreshControls*() method, which is useful if you have set a field's *value* property through code.

The *dataLink* property is similar to the *dataSource* property used for Image objects, except that data displayed through the *dataLink* property can be changed, while data displayed through the *dataSource* property is always read-only.

A component's *dataLink* is automatically set when you use the Form Expert or use a field in the Field Palette.

dataSource

Related topics Example

The bitmap that is displayed in an Image object.

Property of

Image

Description

An Image object can display either a static file from disk or a bitmap stored in a table. Set the *dataSource* property to either one of the following:

- A string containing the word FILENAME, a space, and the name of a file. The string is not case-sensitive.
- A reference to a field object in an open query that contains bitmapped images.

If you assign a field object as the *dataSource*, the Image object will automatically update as you navigate from row to row, unless the rowset's *notifyControls* property is set to *false*.

The *dataSource* property is similar to the *dataLink* property used for Field objects, except that data displayed through the *dataLink* property can be changed, while data displayed through the *dataSource* property is always read-only.

An Image object's *dataSource* is automatically set when you use the Form Expert or use a bitmap image field in the Field Palette.

dataSource example

The following string would set the *dataSource* of an Image object to the file LOGO.GIF in the current directory:

```
filename LOGO.GIF
```

elements

Related topics Example

An array containing object references to all the components in a form.

Property of

Form

Description

The *elements* array contains an object reference for each component in a form.

You can determine the number of components in the form by checking the *elements* array's *length* property. Each element in the array can be addressed by its element number or by the *name* of the component.

The *elements* array is not a member of the Array class, but rather an ObjectArray class with specific capabilities for managing a list of objects. It does not support most of the methods of the Array class. The *elements* array is not meant to be changed directly, although it is safe to scan to get the object references for the components in the form.

A form's *elements* array contains more objects client-side than server-side, because of additional Hidden components that are automatically created by the IntraBuilder Server when the form is rendered into HTML. Therefore, the element number of an object is not the same on the client as it is on the server. When referring to a specific component, use its *name* instead. In most cases, it's easier to use the *name* as a property of the form object than as an element in the *elements* array. For example, the following two expressions refer to the same property:

```
this.form.elements[ "text1" ].value
this.form.text1.value                // This one is shorter and more direct
```

elements example

The following server-side form method checks the *className* property of each component in the *elements* array to find all the CheckBox objects and sets their *checked* properties to *false*.

```
function uncheckAll()
{
  for ( var nElement = 0; nElement < this.elements.length; nElement++ ) {
    if ( this.elements[ nElement ].className == "CheckBox" ) {
      this.elements[ nElement ].checked = false;
    }
  }
}
```

focus()

Related topics

Client-side method: sets focus to a component.

Syntax

<oRef>.*focus*()

<oRef>

A reference to the object to receive focus.

Property of

ListBox, Password, Select, Text, TextArea

Description

Calling a component's *focus*() method sets focus to that component. For example, you can create an *onClick* event for a check box so that when the box is checked, focus automatically moves to a particular Text object.

Although it's intended as a client-side method, *focus*() works when the form is run in the IntraBuilder Designer.

fontBold

Related topics

Specifies whether the component displays its text in bold type.

Property of

CheckBox, HTML, Radio

Description

Set *fontBold* to *true* if you want the component to display its text in boldface.

fontItalic

Related topics

Specifies whether the component displays its text in italics.

Property of

CheckBox, HTML, Radio

Description

Set *fontItalic* to *true* if you want the component to display its text in italics.

fontName

Related topics

The typeface of the component's text.

Property of

CheckBox, HTML, Radio

Description

Set *fontName* to the name of the typeface you want to apply to the text in the component.

fontStrikeout

Related topics

Specifies whether the component displays its text struck through.

Property of

CheckBox, HTML, Radio

Description

Set *fontStrikeout* to *true* if you want the component to display its text struck through.

fontUnderline

Related topics

Specifies whether the component displays its text underlined.

Property of

CheckBox, HTML, Radio

Description

Set *fontUnderline* to *true* if you want the component to display its text underlined.

form

Related topics Example

The form or report that contains the component.

Property of

All form components: ActiveX, Button, CheckBox, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

A component's *form* property is a reference to the form or report that contains it. It is set automatically when the component is created and cannot be changed.

Use the *form* property in component event handlers and methods to generically refer to the object that contains the component.

In a form, a component's *form* and *parent* property refer to the same thing—the form—but in a report, a component is contained deeper in the object hierarchy and its parent is not the report.

By using the *form* property, you can immediately get back to the top of the object hierarchy and refer to its properties, events, or methods; or refer to other objects in the form or report.

form example

The following is an *onServerClick* event handler for a button that puts the form's primary rowset—a data access component—in Append mode to add a new row.

```
function addButton_onServerClick()
{
  this.form.rowset.beginAppend(); // Use form property to get to other
objects in form
}
```

gridLineWidth

Related topics

The width of the HTML table grid lines when the form is displayed on the browser.

Property of

Form

Description

IntraBuilder uses HTML tables to line up components in the browser.

In a form, *gridLineWidth* defaults to zero, so the user does not see grid lines for the HTML table. Set *gridLineWidth* to 1 or higher to make the grid lines visible.

In a report, the PageTemplate's *gridLineWidth* defaults to 1.

groupName

Example

The name of a group of radio buttons.

Property of

Radio

Description

Radio buttons must be used in groups of two or more. Only one radio button in the group may be selected at any time.

To create a group of radio buttons, assign the same string to each Radio object's *groupName* property. The value of the string doesn't really matter; it has no other function besides grouping the radio buttons, and is not related to any other property.

groupName example

Suppose you're creating an order entry screen. For the options "Cash, Check, or Charge," you use three Radio objects with the *groupName* "payment". For the options "Phone, Fax, or E-mail," you use three other Radio objects with the *groupName* "orderedBy".

headTag

Related topics Example

Extra tags to include in the <HEAD> section of the document.

Property of

Form

Description

Use the *headTag* property to include extra tags in the <HEAD> section of the HTML document generated by the IntraBuilder Agent. The contents of the *headTag* property are included as is, just before the </HEAD> tag.

Tags must be enclosed in angle brackets.

headTag example

You can instruct most browsers to reload the document automatically after a certain number of seconds by including a <META HTTP-EQUIV=”REFRESH”> tag in the document head. To include this tag in a document generated by IntraBuilder, set the *headTag* property to something like this:

```
<meta http-equiv="refresh" content="15">
```

height

Related topics

The height of an object.

Property of

Form components: ActiveX, CheckBox, Form, HTML, Image, JavaApplet, ListBox, Radio, TextArea; report components: Band, PageTemplate, StreamFrame

Description

IntraBuilder forms are measured in characters, using an averaged height and width for 8-point MS Sans Serif characters, which is the default font. Therefore sizes on forms roughly correspond to the number of characters that fit in that space.

One unit of form height is approximately three times as large as a unit of width.

IntraBuilder reports are measured in twips (20th of a point). There are exactly 1440 twips per inch.

The *height* of a component is strictly observed in the IntraBuilder Designer, but when the form or report is rendered as HTML, the resulting height may be approximated. The *height* property of a form has no effect when the form is rendered as HTML.

left

Related topics

The position of the left edge of an object relative to its container.

Property of

All form objects: ActiveX, Button, CheckBox, Form, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

IntraBuilder forms are measured in characters, using an averaged height and width for 8-point MS Sans Serif characters, which is the default font. Therefore sizes on forms roughly correspond to the number of characters that fit in that space.

One unit of height is approximately three times as large as a unit of width.

An object's *left* property contains the location of its left edge, relative to the object's container. For form components, the container is the form. For forms, the container is the IntraBuilder Designer itself.

The *left* edge of a component is strictly observed in the IntraBuilder Designer, but when the form is rendered as HTML, it is used only for the relative horizontal alignment of the components in the browser. For components that should be aligned on the left, make sure their *left* properties are the same. You can do this in the Inspector or with the Form Designer's alignment tools.

The *left* property of a form has no effect when the form is rendered as HTML.

linkColor

Related topics

The color of hyperlinks.

Property of

Form

Description

The *linkColor* property determines the color of the hyperlinks. Hyperlinks are created with <A HREF> tags in HTML components.

The value of *linkColor* may be any hexadecimal RGB (Red Green Blue) triplet or a JavaScript color name, as listed under the *color* property.

Use the *vlinkColor* property for the color of hyperlinks to documents that have been visited.

move()

Related topics Example

Repositions and resizes an Image object.

Syntax

<oRef>.*move*(<left expN> [, <top expN> [, <width expN> [, <height expN>]]])

<oRef>

The Image object to move or resize.

<left expN>

The new *left* property.

<top expN>

The new *top* property.

<width expN>

The new *width* property. To change the size of the image, you must specify both the <left expN> and the <top expN>.

<height expN>

The new *height* property.

Property of

Image

Description

Use *move*() to reposition and/or resize an Image object in one step. You could assign the four properties directly, but doing so would require four separate steps, and the image would have to be moved and/or resized after each step. Using *move*() is faster.

If you want to resize the Image object without moving it, pass the current *left* and *top* properties as parameters to *move*(), along with the new width and height.

If you're using *move*() to resize an image, the object's *alignment* property should be set to either Stretch (0) or Keep Aspect Stretch (3).

move() example

The following are two *onServerClick* event handlers for buttons that zoom and unzoom a bitmap image.

```
function zoomButton_onServerClick()
{
  var map = this.form.mapImage; // Store object reference in variable to
reduce typing
  map.move( map.left, map.height, 60, 20 );
}
function unzoomButton_onServerClick()
{
  var map = this.form.mapImage; // Store object reference in variable to
reduce typing
  map.move( map.left, map.height, 30, 10 );
}
```

multiple

Related topics

Specifies whether a ListBox object allows selection of more than one item at a time.

Property of

ListBox

Description

Set *multiple* to *true* if you want to allow the selection of more than one item at one time in a ListBox object.

The selections—whether there's one, many, or none—are stored in the ListBox object's *selected* array.

name

Related topics

The name of the form property that is used to refer to a component.

Property of

All form components: ActiveX, Button, CheckBox, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

A component's *name* property reflects the name of the property of the form that is used to refer to the component.

For example, if pushing one button makes another button visible, the code looks like this:

```
function oneButton_onServerClick()
{
  this.form.anotherButton.visible = true;
}
```

In oneButton's event handler, *this* refers to the button itself, *form* refers to the form that contains the button, and anotherButton is a property of the form that contains an object reference to the Button object anotherButton.

When the form was created in the Form Designer, the *name* property of the Button object was set to anotherButton. When the form is saved into a JFM file, the resulting JavaScript code for the button looks like this:

```
with (this.anotherButton = new Button(this)) {
  left = 10;
  top = 0;
  width = 8;
}
```

The name of the button is never assigned to the *name* property. Instead, the name of the button is determined by the name of the form property that contains the reference to the object. This is true for any form component that has a *name* property.

To change the name of a component in the JFM file, change the name of the property in the *with* statement.

When you read a component's *name* property, IntraBuilder returns the name of the property that the component's *parent* (the form) uses to refer to the object.

If you assign a value to a component's *name* property, you actually change the name of the form property that contains the component's object reference. While this is allowed, there aren't many reasons you would want to do that—avoid it.

onBlur

Related topics Example

Client-side event: when a component loses focus.

Parameters

none

Property of

ListBox, Select, Text, TextArea

Description

onBlur fires whenever the component loses focus. Although it's intended as a client-side event, *onBlur* will fire under the same conditions when the form is run in the IntraBuilder Designer.

Unlike *onChange*, which fires only if the value or selection in a component has changed, *onBlur* always fires when a component loses focus.You can use *onBlur* to make sure something is filled in and not blank.

If there is no other component on the form that can receive focus, for example your form has just one Text component and an image map, the component will never lose focus, so *onBlur* will never fire while the user stays inside the form.

All *onBlur* event handlers created in IntraBuilder are automatically exported in HTML as client-side JavaScript.

onBlur fires after *onChange*, so you should not duplicate actions if you're using both event handlers.

onBlur example

The following *onBlur* event handler executes on the client browser and uses the isBlank() function to tell the user if they've left the Text object blank—either empty of just full of spaces.

```
function isBlank( cArg )
{
// {Export} This comment causes this function body to be sent to the client
  // Trim all trailing blanks
  while ( cArg.length > 0 && cArg.charAt( cArg.length - 1 ) == " " ) {
    cArg = cArg.substring( 0, cArg.length - 1 );
  }
  // If nothing's left (or there was nothing to begin with), it's blank
  return cArg == "";
}
function requiredText_onBlur()
{
  if ( isBlank( this.value ) ) {
    alert( "Don't leave the field blank" );
  }
}
```

onChange

Related topics Example

Client-side event: when a component loses focus, and the contents of the component have been changed.

Parameters

none

Property of

ListBox, Select, Text, TextArea

Description

Use *onChange* to validate data entered into a form. *onChange* fires when the component loses focus, if the value or selection in the component has changed. Although it's intended as a client-side event, *onChange* will fire under the same conditions when the form is run in the IntraBuilder Designer.

If the value or selection in the component has not changed, *onChange* does not fire. Therefore *onChange* is not suitable for making sure something is filled in and not blank (if the value of the component is blank to begin with). Use *onBlur* instead.

If there is no other component on the form that can receive focus (for example, if your form has just one Text component and an image map), the component will never lose focus, so *onChange* will never fire while the user stays inside the form.

All *onChange* event handlers created in IntraBuilder are automatically exported in HTML as client-side JavaScript.

onBlur fires after *onChange*, so you should not duplicate actions if you're using both event handlers.

onChange example

Suppose you're writing an online ticket ordering system that does not allow someone to order more than six tickets at once. The following *onChange* event handler executes on the client browser to tell the user if they've tried to order too many tickets, before the user submits the form.

```
function numTickets_onChange()
{
  if ( parseInt( this.value ) > 6 ) {
    alert( "You may order a maximum of six tickets at a time" );
  }
}
```

onClick

Related topics Example

Client-side event: when a component is clicked.

Parameters

none

Property of

CheckBox, Button, Reset

Description

Use *onClick* to execute code on the client browser when you click a component.

If you set an *onClick* event for a Button object, any *onServerClick* event that you have set for the button will not fire when the button is clicked in the browser.

If you want the Button object to do something on both the client and server, set the *onClick* event to perform the action on the client, then call the form's client-side *submit*() method. This will in turn cause the form's *onServerSubmit* event (not the button's *onServerClick* event) to fire, and you can perform your server-side action from the *onServerSubmit* event handler.

Although it's intended as a client-side event, *onClick* will fire when the form is run in the IntraBuilder Designer. If you have both an *onClick* and *onServerClick* event for a button, they will both fire (in that order) when the form is run in the IntraBuilder Designer.

All *onClick* event handlers created in IntraBuilder are automatically exported in HTML as client-side JavaScript.

onClick example

The following *onClick* handler calls a function to launch a separate status window using client-side JavaScript, then submits the form so that the desired action will begin.

```
function startButton_onClick()
{
  launchStatusWin();
  this.form.submit();  // Submit form with client-side JavaScript
}
function launchStatusWin()
{
// {Export} This comment causes this function body to be sent to the client
  // Create status window....
}
```

onDesignLoad

After a form or component is loaded in the Form Designer.

Parameters

<from palette expL>

Whether the component was added from the palette. If *true*, the component has just been created. If *false*, the component has been reloaded into the Form Designer (when editing an existing form).

Property of

All form objects: ActiveX, Button, CheckBox, Form, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

Use *onDesignLoad* to execute code whenever a form or component is loaded into the Form Designer, either when it is first created (for components only), or when it is subsequently loaded into the Form Designer.

onFocus

Related topics Example

Client-side event: when a component gains focus.

Parameters

none

Property of

ListBox, Select, Text, TextArea

Description

onFocus fires whenever the component gains focus. Although it's intended as a client-side event, *onFocus* will fire under the same conditions when the form is run in the IntraBuilder Designer.

Be careful not to do something in an *onFocus* event handler that causes the component to temporarily lose and regain focus. For example, if you display a message with *alert*(), the component loses focus, then, when the user dismisses the dialog box, the component regains focus and the *onFocus* event fires again, causing an infinite loop. The same thing can happen with the browser's error dialog if you have a run-time error in your *onFocus* event handler.

All *onFocus* event handlers created in IntraBuilder are automatically exported in HTML as client-side JavaScript.

onFocus example

The following *onFocus* event handler keeps track of the number of times the user has visited each component by incrementing a number in the Hidden object hiddenCounter. All the components on the form use the same *onFocus* event handler to track usage patterns.

```
function countFocus()
{
  this.form.hiddenCounter.value++;
}
```

onImageClick

Related topics Example

Client-side event: when an Image object is clicked.

Parameters

none

Property of

Image

Description

Use *onImageClick* to execute code on the client browser when you click an Image object. Unlike other client-side events, *onImageClick* does not fire in the IntraBuilder Designer.

If you set an *onImageServerClick* event for an Image object, any *onImageClick* event that you have set for the image will not fire when the image is clicked in the browser.

onImageClick does not receive any parameters, so you cannot use *onImageClick* to implement an image map. Use *onImageServerClick* instead.

onImageClick example

The following *onImageClick* handler simulates the action of a reset button, clearing component values.

```
function startOver_onImageClick()
{
  var f = document.forms[ 0 ];  // Get reference to form
  f.firstName.value = "" ;
  f.lastName.value  = "" ;
}
```

onImageServerClick

Related topics Example

After an Image object is clicked.

Parameters

<left expN>

The horizontal coordinate of the pixel that was clicked, from the left edge of the image.

<top expN>

The vertical coordinate of the pixel that was clicked, from the top edge of the image.

Property of

Image

Description

Use *onImageServerClick* to execute code when you click an Image object.

If you set an *onImageServerClick* event for an Image object, any *onImageClick* event that you have set for the image will not fire when the image is clicked in the browser.

The *onImageServerClick* event handler receives the coordinates of the pixel in the image that was clicked. You can use these values to implement an image map, where clicking on different parts of the image cause different actions to occur.

Set the *alignment* property of the Image object to True Size to prevent IntraBuilder from dynamically resizing the image, so that you can accurately determine which part of the image was clicked.

onImageServerClick example

Suppose you have a horizontal VCR-style navigation bar, with button images for First, Previous, Next, and Last row. Use the following *onImageServerClick* event handler to take the appropriate action.

```
function navBar_onImageServerClick( nLeft, nTop )
{
  #define BUTTON_WIDTH 26
  #define FIRST_BUTTON  0
  #define PREV_BUTTON   1
  #define NEXT_BUTTON   2
  #define LAST_BUTTON   3
  // Divide horizontal pixel value by button width and round down to figure
out
  // which image was clicked
  var nBtn = Math.floor( nLeft / BUTTON_WIDTH );
  switch ( nBtn ) {
    case FIRST_BUTTON:
      form.rowset.first();
      break;
    case PREV_BUTTON:
      if ( !form.rowset.next( -1 ) ) {
        form.rowset.next();
      }
      break;
    case NEXT_BUTTON:
      if ( !form.rowset.next() ) {
        form.rowset.next(-1);
      }
      break;
    case LAST_BUTTON:
      form.rowset.last();
      break;
  }
}
```

onLoad

Related topics

Client-side event: after the entire document has been loaded.

Parameters

none

Property of

Form

Description

onLoad fires after the entire document has been loaded (after all the components in the form have been loaded into the browser).

A form's *onServerLoad* event fires only once, when the form is first loaded by the IntraBuilder Agent. When the user submits the form (for example navigates to another row), *onServerLoad* does not fire again, because the form has been loaded only once in the IntraBuilder Agent. In contrast, the form's *preRender* event is fired on the server every time the form is rendered (after the form first loads or in response to a submit) and transmitted. Once the browser has finished receiving and loading the transmitted form, *onLoad* also fires.

All *onLoad* event handlers created in IntraBuilder are automatically exported in HTML as client-side JavaScript.

onSelect

Related topics

Client-side event: when text in a component is selected.

Parameters

none

Property of

Text, TextArea

Description

onSelect fires whenever text in a Text or TextArea is selected.

All *onSelect* event handlers created in IntraBuilder are automatically exported in HTML as client-side JavaScript.

onServerClick

Related topics Example

After a button is clicked.

Parameters

none

Property of

Button

Description

Use *onServerClick* to execute code when you click a button. Unless a Button object has an *onClick* event handler, all buttons in IntraBuilder are HTML submit buttons. Clicking on a button submits the form so that actions can take place on the server; the updated form is transmitted back to the browser.

If you set an *onClick* event for a Button object, any *onServerClick* event that you have set for the button will not fire when the button is clicked in the browser. If you have both an *onClick* and *onServerClick* event, then they will both fire (in that order) when the form is run in the IntraBuilder Designer.

If you want the Button object to do something on both the client and server, set the *onClick* event to perform the action on the client, then call the form's client-side *submit*() method. This will in turn cause the form's *onServerSubmit* event (not the button's *onServerClick* event) to fire, and you can perform your server-side action from the *onServerSubmit* event handler.

onServerClick example

The following *onServerClick* event handler for an Add button puts the form's primary rowset in Append mode to allow entry of a new row.

```
function newButton_onServerClick()
{
  this.form.rowset.beginAppend();
}
```

onServerLoad

Related topics Example

After the form or component has been opened.

Parameters

none

Property of

All form objects: ActiveX, Button, CheckBox, Form, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

onServerLoad events fire after a form has been opened. First the *onServerLoad* event for the form fires, then the *onServerLoad* for each component, if one has been assigned. Use *onServerLoad* to set up items in the form before it is first transmitted to the browser.

A form's *onServerLoad* event fires only once, when the form is first loaded by the IntraBuilder Agent. When the user submits the form (for example navigates to another row), *onServerLoad* does not fire again, because the form has been loaded only once in the IntraBuilder Agent. In contrast, the form's *preRender* event is fired on the server every time the form is rendered (after the form first loads or in response to a submit) and transmitted. Once the browser has finished receiving and loading the transmitted form, *onLoad* also fires.

onServerLoad example

The following *onServerLoad* event handler for the form calls the form's refreshUnlinked() method to update components on the form that are not *dataLink*ed directly to fields so that the components contain the correct information when the form is first transmitted to the browser.

```
function newButton_onServerClick()
{
  this.refreshUnlinked();
}
function refreshUnlinked()
{
  // Update unlinked components....
}
```

onServerSubmit

Related topics Example

After the form is submitted by the client-side *submit*() method.

Parameters

none

Property of

Form

Description

Forms are typically submitted by clicking on a button or image. Once the form is submitted, the corresponding *onServerClick* or *onImageServerClick* event is fired so that actions can take place on the server; the updated form is transmitted back to the browser.

You can also submit a form by calling its *submit*() method with client-side JavaScript. The primary reason to do this is if you want a button (or image) to do something on both the client and server. In that case, set the *onClick* (or *onImageClick*) event to perform the action on the client, then call the form's client-side *submit*() method. This will in turn cause the form's *onServerSubmit* event (not the button's *onServerClick* event) to fire, and you can perform your server-side action from the *onServerSubmit* event handler.

If you have multiple buttons on a form that call the form's client-side *submit*(), you can store a value in a Hidden object beforehand so that the *onServerSubmit* event handler can detect which button was clicked.

On some browsers, pressing the Enter key when no button has focus also submits the form, which would also fire the form's *onServerSubmit*.

onServerSubmit example

Suppose you have two buttons in a form in one frame of a two-frame HTML frameset. You want one button to launch a form in the current frame, and the other button to display a report in the second frame. To direct output to the correct frame, you need to set the form's *target* property client-side before submitting the form.

The following code contains two *onClick* event handlers for the buttons, which set a value in a Hidden component and submit the form, and the *onServerSubmit* event handler, which checks the Hidden component to determine which action to take.

IntraBuilder tracks submitted forms with an internally generated sequence number stored in a Hidden component that is always the second element in the form (element number 1). Because this example directs output to another frame, the same form may be submitted more than once. Therefore, the sequence number must be set to a special value, –1, to force the IntraBuilder Agent to accept the form even if it has already been submitted before.

```
function viewerButton_onClick()
{
  this.form.target = "_self";                // Output to current frame
  this.form.hiddenAction.value = "VIEWER";   // Run "VIEWER" form
  this.form.elements[ 1 ].value = "-1";      // Force Agent to accept reposted
form
  this.form.submit();
}
function listNewButton_onClick()
{
  this.form.target = "reportFrame";          // Output to other frame
  this.form.hiddenAction.value = "NEW";      // Run "NEW" report
  this.form.elements[ 1 ].value = "-1";      // Force Agent to accept reposted
form
  this.form.submit();
}
function Form_onServerSubmit()
{
  if ( this.hiddenAction.value == "VIEWER" ) {
    _sys.forms.run( "VIEWER");
  }
  else if ( this.hiddenAction.value == "NEW" ) {
    _sys.reports.run( "NEWMSG" );
  }
}
```

onServerUnload

Related topics

After the form has been closed.

Parameters

none

Property of

Form

Description

Use *onServerUnload* to perform any extra manual cleanup, if necessary, when you close a form. Normally, IntraBuilder will automatically discard anything in the form when you close it. You might use *onServerUnload* if you created an object in the *onServerLoad* that you did not bind to the form.

Unless you explicitly call a form's *close*() method, forms do not close unless they time-out or the same user reloads the same form, in which case the old version is discarded. If the user simply leaves a form alone (for example by going to another location on the Web or closing the browser), you don't know that the user is done until the form times-out. In that case, the user did not complete their task; they were simply terminated. Therefore you cannot rely on the closing of a form to signify the normal completion of an action.

open()

Related topics Example

Opens a form.

Syntax

<oRef>.*open*()

<oRef>

An object reference to the form you want to open.

Property of

Form

Description

Use *open*() to open a form.

When running on the IntraBuilder Agent, opening a form places it on the top of the form stack. The form on the top of the form stack is the one that is transmitted to the client browser.

The standard bootstrap code opens an instance of the form when you run the form's JFM script file, so to open a form, run the JFM with ***_sys.forms.run***() or ***_sys.scripts.run***().

After a form has been opened, its *onServerLoad* events fire.

open() example

The following is an *onServerClick* event handler for a button that launches another form.

```
function viewerButton_onServerClick()
{
  _sys.forms.run( "VIEWER" );  // Launch the Viewer form
}
```

The first executable statements at the top of the JFM script file comprise the standard bootstrap code, which creates an instance of the form and calls the new form's *open*() method:

```
var f = new ViewerForm();
f.open();
class ViewerForm extends Form {
Ä
```

options

Related topics Example

The options that are displayed in a Select or ListBox object.

Property of

ListBox, Select

Description

Use the *options* property to set the options that are displayed in a Select or ListBox object. The *options* property is a string in one of the following forms:

- The word FILENAME, a space, and a file mask with * and/or ? wildcard characters. The options would consist of the files in the directory that match the file mask.
- The word ARRAY, a space, and an array; either a literal array or a valid reference to an existing Array object. The options would consists of all the elements in the array.

The file mask and the words FILENAME and ARRAY are not case-sensitive. The contents of a literal array and the reference to an Array object are case-sensitive.

Adding elements to an array after it has been assigned as a component's *options* may not automatically update the component's options. Files added to the directory after the *options* property has been set to a file mask will not automatically appear either.

To update the *options*, you need to reassign the *options* property. In most cases, you can simply reassert the property by assigning its current value to itself. For example, if you had originally specified all the GIF files in the current directory, the *options* property assignment would look like this:

```
with (this.fileSelect = new Select(this)) {
  options = "filename *.GIF";
}
```

To update the file list when you press an Update button on your form, the button's *onServerClick* would look like this:

```
function updateButton_onServerClick()
{
  this.form.fileSelect.options = this.form.fileSelect.options;
}
```

You don't have to specify what the *options* string is again, since it's already contained in the *options* property. This makes your code easier to maintain, since the *options* string is specified in only one place.

When using an array in the *options* string, you can use a literal array, for example,

```
array {"Chocolate", "Strawberry", "Vanilla"}
```

Or you can use a reference to an array object, for example,

```
array aFlavors
```

If you use a reference, that array must exist at the time the *options* property is assigned. Since the *options* property contains that string (in this example, array aFlavors), if you reassert the *options* property as shown above, an updated version of the named array must exist. In this example, the array aFlavors must be accessible in the method updateButton_onServerClick().

For this reason, when using an updatable array as the *options* property, the array is usually created as a property of the form. This makes the array equally accessible from the Select or ListBox component that uses the array and from any other component that tries to reassert the *options* property. In this example, the array aFlavors would be created as a property of the form, and the *options* string would contain:

```
array this.form.aFlavors
```

The reference *this*.*form*.aFlavors is valid from the event handler of any component on the form.

options example

The following *onServerLoad* event handler reads the contents of a field in a table into an array to be used as the Select object's options. A table of ice cream flavors has already been opened in a query named flavors1.

```
function flavorSelect_onServerLoad()
{
  this.form.aFlavors = new Array();
  this.form.flavors1.rowset.first();
  while ( !this.form.flavors1.rowset.endOfSet ) {
    this.form.aFlavors.add( this.form.flavors1.rowset.fields[ "Name" ].value
);
    this.form.flavors1.rowset.next();
  }
  this.options = "array this.form.aFlavors";
}
```

Later, if someone adds a new flavor, they can add it to the array and update the Select object immediately. (The flavor will be added to the table once it's approved by the flavor committee.)

```
function addFlavorButton_onServerClick()
{
  this.form.aFlavors.add( this.form.newFlavorText.value );  // Add new flavor
  this.form.flavorSelect.options = this.form.flavorSelect.options;  // Update
Select
}
```

pageno

Related topics Example

The page of the form on which a component appears, or the form's active page.

Property of

All form objects: ActiveX, Button, CheckBox, Form, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

All form objects have a *pageno* property that can be between 0 and 255. The form's *pageno* property indicates the form's active page, the one it is displaying. All the components in the form that have the same *pageno* as the form are displayed on that "page"; the rest are hidden. There are no actual pages or page objects to manage.

When a form's *pageno* property is zero, all components are displayed. If a component's *pageno* property is zero, it appears on all pages. For example, a company logo that appears on every page can be placed on page zero.

The *pageno* property can be changed at any time. Changing a form's *pageno* displays another page of the form. Changing a component's *pageno* moves that component to that page.

In addition to the *pageno* property, you can set a component's *visible* property if you want to hide or display it under particular circumstances.

pageno example

Suppose you have a 12-page survey form. There are buttons to move to the next and previous pages. These buttons are on page zero, so that they appear on every page. The Previous button has its *visible* property initially set to *false*, because the form starts on page 1 and there is no previous page to go to. When you get to page 12, you want to hide the Next button, since there are no more pages.

The *onServerClick* event handlers for the two buttons would look like:

```
function nextButton_onServerClick()
{
  if ( ++this.form.pageno >= 12 ) {     // Goto next page, and if it's the
last page
    this.visible = false;               // You can't go any further
  }
  this.form.prevButton.visible = true;  // Always make previous button
visible
}
function prevButton_onServerClick()
{
  if ( --this.form.pageno <= 1 ) {  // Goto previous page, and if it's the
first page
    this.visible = false;           // You can't go any further
  }
  this.form.nextButton.visible = true;  // Always make next button visible
}
```

The prefix increment and decrement operators are used so that the page number is changed before it is tested. It's not necessary to see if you should be allowed to change pages; if the button is visible, you can go in that direction. Finally, going in one direction always makes it possible to go the other way.

params

Related topics Example

Parameters passed to a Java applet or ActiveX control.

Property of

ActiveX, JavaApplet

Description

The *params* property contains an associative array that contains parameter names and values, if any, that are passed to the Java applet or ActiveX control.

params example

The following is a code excerpt from a JFM form file that creates an ActiveX component that defines the control's *classId*, *codeBase*, and a number of parameters in the *params* array:

```
with (this.activex1 = new ActiveX(this)) {
   height = 10;
   left = 4;
   top = 1;
   width = 30;
   classId = "clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2";
   codeBase = "http://activex.microsoft.com/controls/iexplorer/ielabel.ocx";
   params["Angle"]="45";
   params["Alignment"]="4";
   params["FontBold"]="1";
   params["ForeColor"]="#ffa500";
   params["FontName"]="Arial";
   params["BackStyle"]="0";
   params["Caption"]="Welcome!";
   params["FontSize"]="20";
}
```

preRender

Related topics Example

Before the form or report is rendered.

Parameters

none

Property of

Form

Description

preRender is fired every time the form is transmitted to the browser, just before it is rendered into HTML. Use it for items on the form that need to be updated every time it is transmitted; that is, after every action that submits the form.

A form's *onServerLoad* event fires only once, when the form is first loaded by the IntraBuilder Agent. When the user submits the form (for example navigates to another row), *onServerLoad* does not fire again, because the form has been loaded only once in the IntraBuilder Agent. In contrast, the form's *preRender* event is fired on the server every time the form is rendered (after the form first loads or in response to a submit) and transmitted. Once the browser has finished receiving and loading the transmitted form, *onLoad* also fires.

preRender example

The following is a *preRender* event handler for a form that displays the HTML object minClock as a minute clock. The *preRender* event fires every time the form is rendered, so the clock is always updated. The clock displays the minutes and seconds only, because the page is designed to be seen in different time zones and all you want to display is the number of minutes (and seconds) past the hour.

```
function Form_preRender()
{
  var dNow = new Date();           // Get time once
  var xMin = dNow.getMinutes();    // Get minutes
  var xSec = dNow.getSeconds();    // and seconds
  if ( xMin < 10 ) {               // Add leading zero if needed
    xMin = "0" + xMin;
  }
  if ( xSec < 10 ) {
    xSec = "0" + xSec;
  }
  this.minClock.text = xMin + ":" + xSec;  // Change text
}
```

readOnly

Specifies whether changes to the component are saved into the *dataLink*ed field.

Property of

TextArea

Description

There are two ways to display read-only versions of data:

- Place the rowset in Browse mode, so that IntraBuilder automatically generates static text instead of data entry components.
- Create and update unlinked HTML components yourself.

The TextArea component is the exception. Because it's intended for displaying multiple lines of text, it's not always feasible to create a static text version of the contents. For example, if it contained a hundred lines of text, that would push everything after it far down in the page. Therefore, the read-only version of a TextArea component is also a TextArea component.

If the rowset is in Browse mode, the data in the rowset is automatically read-only. If you make any changes in the TextArea component, they are ignored.

You can also set the TextArea's *readOnly* property to *true* if you want to make that component alone read-only. For example, you could display the content of a technical paper that is not editable, but allow changes to other fields, such as the name of the paper's reviewer. Again, the user will actually be able to make changes in the TextArea component in their browser, because HTML does not support read-only components (yet), but any changes the user makes will be ignored.

right

Related topics

The position of the right edge of an object relative to its container.

Property of

Rule

Description

IntraBuilder forms are measured in characters, using an averaged height and width for 8-point MS Sans Serif characters, which is the default font. Therefore sizes on forms roughly correspond to the number of characters that fit in that space.

One unit of height is approximately three times as large as a unit of width.

A Rule object does not have a *width*; the name of the property can be confused with line thickness. Instead, a Rule has both a *left* and *right* property that determines the length of the line.

The *right* edge of an object is strictly observed in the IntraBuilder Designer, but when the form is rendered as HTML, it is used only for the relative horizontal sizing of the components in the browser.

selected

Related topics Example

An Array object containing the items marked as selected in a component.

Property of

ListBox

Description

By setting a ListBox object's *multiple* to true, you can select more than one item at a time. The *selected* property contains a reference to an Array object that contains the ListBox object's currently selected items, one element per selection.

As with any Array object, check its *length* property to determine how many items have been selected. The items in the *selected* array are always listed in the same order as they are in the ListBox.

The ListBox object's *value* property contains the value of the selection that currently has focus, whether it's selected or not.

selected example

The following *onServerClick* event handler copies all the selected items in the ListBox object select1 into another ListBox object named select2:

```
function makeSelections_onServerClick()
{
  this.form.select2.options = "array this.form.select1.selected";
}
```

size

Related topics

The thickness of a Rule object.

Property of

Rule

Description

A Rule object's *size* property dictates the thickness of the line, in pixels.

submit()

Related topics Example

Submits the form.

Syntax

<oRef>.*submit*()

<oRef>

The Form object to submit.

Property of

Form

Description

There are three ways to submit a form from a browser:

- Assign an *onServerClick* event to a Button object. Clicking the button submits the form and calls that *onServerClick* event handler.
- Assign an *onImageServerClick* event to an Image object. Clicking the image submits the form and calls that *onImageServerClick* event handler.
- Call the form's *submit*() method from any client-side function, such as an *onClick* event handler. This submits the form and calls the form's *onServerSubmit* event handler.

submit() is supported server-side in the IntraBuilder Designer to facilitate testing. It triggers the form's *onServerSubmit* event.

submit() example

The following client-side *onClick* event handler assigns some values to objects in the browser, then submits the form by calling the form's *submit*() method.

```
function listNewButton_onClick()
{
  this.form.target = "reportFrame";
  this.form.hiddenAction.value = "NEW";
  this.form.elements[ 1 ].value = "-1";
  this.form.submit();
}
```

template

Example

A formatting template for text.

Property of

HTML, Text

Description

Set the *template* property to a format text. A template is a string that consists of template characters, which represent and modify individual characters in the text string, and literal characters, which are inserted into the text.

The supported template characters are:

Character	Description
X	Allows any character and displays it unchanged
!	Allows any character and displays it in uppercase
A	Allows only alphabetic characters
#	Allows digits, spaces, + and –
9	Allows only digits
.	Inserts decimal point
,	Inserts thousands separator

If the data is longer than the length of the *template* string, it is truncated to match.

When displaying a calculated or morphed field, use a *template* that represents the field's maximum size.

template example

To display the contents of a length 20 field in all uppercase and force all entries to uppercase, use a template with 20 “!” characters:

```
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
```

Suppose you’re using a morphed field that stores an ID number but displays a name. The name can be a maximum of 30 characters, so you set the *template* property of Text component that displays the name to 30 “X” characters:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

text

Related topics Example

The non-editable text that appears in a component.

Property of

CheckBox, Radio, Text

Description

The *text* of a CheckBox or Radio object is the descriptive text that appears beside the actual check box or radio button.

The *text* of an HTML object is the content of the object: the actual HTML text that is transmitted when the form is sent to the browser.

The *text* property of an object may contain any HTML text. If you include HTML tags, they will be sent, but you must be aware of the tags' effects, in relation to other HTML that is automatically generated by IntraBuilder. For example, it's usually safe to include <B> tags to bold text, but it's a bad idea to have </TABLE> or </BODY>, which may preemptively terminate your IntraBuilder form.

You may assign any of the following types of data to the *text* property of an HTML component:

- Boolean
- Numeric
- Integer
- Character
- Object
- Null
- DateTime
- Codeblock

If you assign a codeblock to the *text* property, it must return a value. Use either an expression codeblock or a statement codeblock that uses *return* to return a value. The codeblock is evaluated whenever it is rendered. When a form is run in the IntraBuilder Designer, it is rendered only once when the form is opened. When a form is run from a browser, it is rendered into HTML every time the form is transmitted; that is, when the form is first opened and after every submit.

While *text* is a programmable property in IntraBuilder, there is no *text* property in client-side JavaScript; the *text* of the object has already been rendered into static text.

text example

Setting the *text* property of an HTML component to the following expression codeblock:

```
{||new Date()}
```

causes the current date and time to be displayed in the component every time the form is rendered.

title

Related topics

The title of a form.

Property of

Form

Description

The *title* property contains the title of the form. It is displayed in the title bar of the browser, or in the form itself when the form is run in the IntraBuilder Designer.

top

Related topics

The position of the top edge of an object relative to its container.

Property of

All form objects: ActiveX, Button, CheckBox, Form, Hidden, HTML, Image, JavaApplet, ListBox, Password, Radio, Reset, Rule, Select, Text, TextArea

Description

IntraBuilder forms are measured in characters, using an averaged height and width for 8-point MS Sans Serif characters, which is the default font. Therefore sizes on forms roughly correspond to the number of characters that fit in that space.

One unit of height is approximately three times as large as a unit of width.

An object's *top* property contains the location of its top edge, relative to the object's container. For form components, the container is the form. For forms, the container is the IntraBuilder Designer itself.

The *top* edge of a component is strictly observed in the IntraBuilder Designer, but when the form is rendered as HTML, it is used only for the relative vertical alignment of the components in the browser. For components that should be aligned on the top, make sure their *top* properties are the same. You can do this in the Inspector of JFM code, or with the Form Designer's alignment tools.

The *top* property of a form has no effect when the form is rendered as HTML.

value

Related topics

The component's current value.

Property of

Hidden, ListBox, Radio, Password, Select, Text, TextArea

Description

A component's *value* property reflects its value, which is

- The value that is displayed in a Text, TextArea, or Select component
- Obscured in a Password component, although its true value is maintained
- Whether a Radio component is the one in its group that is selected
- The item that has focus in a ListBox component
- The value that's passed back and forth between the IntraBuilder Agent and the client browser in a Hidden component

Both field and component objects have a *value* property. (For a CheckBox object, its value is its *checked* property, but the concept is the same.) When they are *dataLink*ed, changes in one object's *value* property are echoed in the other. The form component's *value* property reflects the value displayed in the component at any given moment. If the component's value is changed, it is copied into the field, either after the component loses focus (if you're running the form locally in the IntraBuilder Designer) or when the entire form is submitted (if you're running the form remotely on a browser).

The *value* property for all fields in a rowset are set when you first open a query and updated as you navigate from row to row. The *value* properties for components *dataLink*ed to those fields are also updated at the same time, unless the rowset's *notifyControls* property is set to *false*. You can also force the components to be updated by calling the rowset's *refreshControls*() method, which is useful if you have set a field's *value* property through code.

When reading or writing values to *dataLink*ed components in server-side code, you can use the *value* property of either the visual component or the field object; there's no difference, although you should be consistent. You may choose to program the visual interface, if the underlying data is more likely to change; or you might choose to work with the data access objects, so you don't have to worry about the names of the form components and whether they're correctly *dataLink*ed. In general, it's easier and more portable for data access object events to access the fields, so you're more likely to assign to the *value* properties of the fields.

If the component is not *dataLink*ed or has no *dataLink* property (like the Hidden object), then you will work with the form component's *value*. The same is true for client-side code, since there are no data access objects in client-side JavaScript.

virtualRoot

The working directory of a form or report.

Property of

Form

Description

Every connection serviced by an IntraBuilder Agent has its own current drive and directory, which is represented by the form or report's *virtualRoot* property. As each client request is serviced, the IntraBuilder Agent switches to that connection's *virtualRoot* directory. Once the request is finished, the IntraBuilder Agent switches back to the IntraBuilder home directory and waits for the next request. This allows forms and reports that are installed in different directories to be serviced by the same IntraBuilder Agent.

At the beginning of any server-side event handler, the current directory is the form's *virtualRoot* directory. All file references are relative to that directory. If you change directories during the event handler, that change will be lost once the event handler is finished.

The *virtualRoot* property is set by default to the directory that contains the form or report. There's usually no need to change it. Setting the *virtualRoot* property also causes IntraBuilder to change to that directory.

visible

Related topics

Specifies whether a component is visible.

Property of

Button, CheckBox, HTML, Image, ListBox, Password, Radio, Reset, Select, Text, TextArea

Description

Use the *visible* property to conditionally hide a component. If a component is not visible, either because its *visible* property is *false* or it's on another page in the form, the component is not rendered into HTML.

Hidden components have no visual representation when the form is run and no *visible* property, but they do have a *pageno* property and obey standard page behavior.

vlinkColor

Related topics

The color of visited hyperlinks.

Property of

Form

Description

The *vlinkColor* property determines the color of the hyperlinks to documents that have been visited. Hyperlinks are created with <A HREF> tags in HTML components.

The value of *vlinkColor* may be any hexadecimal RGB (Red Green Blue) triplet or a JavaScript color name, as listed under the *color* property.

Use the *linkColor* property for the color of hyperlinks to documents that have not been visited.

width

Related topics

The width of an object.

Property of

Form components: ActiveX, Button, CheckBox, Form, HTML, Image, JavaApplet, ListBox, Radio, Reset, Select, Text, TextArea; report components: PageTemplate, StreamFrame

Description

IntraBuilder forms are measured in characters, using an averaged height and width for 8-point MS Sans Serif characters, which is the default font. Therefore sizes on forms roughly correspond to the number of characters that fit in that space.

One unit of form height is approximately three times as large as a unit of width.

The *width* of a Form object is its interior width, not counting the border in the IntraBuilder Designer. A Rule object's horizontal length is determined by its *right* property, and its thickness is dictated by its *size* property.

IntraBuilder reports are measured in twips (20th of a point). There are exactly 1440 twips per inch.

The *width* of a component is strictly observed in the IntraBuilder Designer, but when the form or report is rendered as HTML, the resulting width may be approximated. The *width* property of a form has no effect when the form is rendered as HTML.

Report objects

Report objects generate formatted output from data in tables. The Report Expert and Report Designer allow you to create and modify reports visually. Reports are saved as JavaScript code in a JRP file that you can modify.

All measurements in reports are in twips (20th of a point). There are exactly 1440 twips per inch.

At the top of the report object class hierarchy is the Report class. A Report object acts as a container for four main groups of objects:

- Data access objects, which give access to data in tables
- Query objects
- Database objects
- Session objects

  These objects are created and used the same way they are in forms, except that a report does not have a primary rowset like a form does.

- Report layout objects, which determine the appearance of the page and where data is output, or streamed
- PageTemplate objects
- StreamFrame objects

  A Report object contains one or more PageTemplates, and each PageTemplate usually contains one or more StreamFrames.

- Data stream objects, which read and organize the data from a query's rowset and stream it out to a report's StreamFrame objects
- StreamSource objects
- Band objects
- Group objects

  Each StreamSource object contains a Band object that is assigned to its *detailBand* property. The contents of the *detailBand* are rendered for each row in the rowset. A StreamSource may also have one or more Group objects, which group data and have their own header and footer Band objects.

- Visual components—objects that display the report's data
- HTML objects
- Image objects
- Rule objects
- CheckBox objects
- Radio objects

  These objects are created as properties of a PageTemplate object if they are fixed elements on the page, such as a report's date and page number; otherwise they are properties of a Band object and are used to display data.

The primary method of displaying information in a report is through HTML objects. For text that varies, such as the data from the rowset, the *text* property of the HTML object is set to an expression codeblock, which is evaluated every time the object is rendered. By using an expression in the codeblock that accesses the fields in the rowset, the HTML object displays data from tables.

You may use the other visual components in a report to display static images or images from a table, draw lines, or display table data with check boxes or radio buttons.

Note Visual component objects are used in forms as well as reports, and most of the properties, methods, and events associated with the objects are described in the Form objects series of topics. Some HTML properties used only in reports are described in this series.

A simple report example

Related topics

To get a sense of how everything fits together, imagine a report of students grouped by grade, with the total number of students in each grade.

The report has a query that accesses the table of students, named students1; a StreamSource object, by default named streamSource1, to stream the data from the query; and a PageTemplate object, by default named pageTemplate1, that describes the physical attributes of the page, such as its dimensions, background color, and margins.

pageTemplate1 contains one StreamFrame object, by default named streamFrame1, where the data stream will be rendered. It occupies most of the space inside pageTemplate1's margins. The rest of the space is used by HTML components that display the report title, date, and page.

streamFrame1 has a *streamSource* property that identifies its StreamSource object. It is assigned streamSource1.

streamSource1 has a *rowset* property that identifies the StreamSource object's rowset. It is assigned students1.*rowset*.

students1.*rowset* and streamFrame1 are now linked. To fill streamFrame1 with data, the report engine will traverse students1.*rowset*, from the first row to the last row. But at this point, no data will be displayed, because there are no visual components in any Band objects.

HTML components are assigned to streamSource1.*detailBand*. The *text* properties of these components are expression codeblocks that refer to the *value* properties of the fields of the *rowset* of the StreamSource object. For example, the *text* of the HTML component that displays the student's last name is

```
{||this.form.students1.rowset.field[ "Last name" ].value}
```

When a visual component is placed in a report, its *form* property refers to the report.

To group the data, a Group object, named group1 by default, is assigned to streamSource1. Its *groupBy* property contains the name of the group field, "Grade". The report engine will watch the value of this field in the rowset, that is:

```
students1.rowset.field[ "Grade" ].value
```

and whenever the value of the field changes, a new group begins. Therefore, it's important that the data is sorted by grade. If the report's *autoSort* property is *true*, all of the report's queries will automatically be sorted to match the groups in the StreamSource objects.

group1 has two Band objects of its own: a header band and a footer band, assigned to the *headerBand* and *footerBand* properties respectively. The *headerBand* is currently empty, and the *footerBand* displays the count of the students in that grade.

The Group object's *agCount*() method counts the number of rows in the group. To display that number, the *text* of the HTML component in the *footerBand* is set to the following expression codeblock:

```
{||"Count: " + this.parent.parent.agCount({||
this.parent.rowset.fields["ID"].value})}
```

The expression codeblock concatenates the text label with the return value of the Group object's *agCount*() method. To get to that method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The *agCount*() method expects a code reference as a parameter that it can evaluate. If the return value is not *null*, the count is incremented. The code reference here is another expression codeblock that uses dot operators:

- *this* is the Group object group1.
- group1's *parent* is streamSource1.
- streamSource1's *rowset* is students1.*rowset*, the rowset that the report engine is traversing to fill

streamFrame1.

That's all the objects that go into a report of students, grouped by grade, with the number of students in each grade. There are two final details that are needed to make the report work.

Because a report can have multiple PageTemplate objects, a Report object has a *firstPageTemplate* property that refers to the PageTemplate object to use for the first page. It is assigned pageTemplate1.

Each PageTemplate object has a *nextPageTemplate* property that refers to the PageTemplate object to use when the current page is done. For pageTemplate1, it is assigned a reference to itself. This means that the same page layout is used for every page in the report.

Everything described in this sample report can be handled automatically by the Report Expert. To run the report, call the Report object's *render*() method.

How a report is rendered

Related topics

When a Report object's *render*() method is called, the first thing the report does is call its *preRender* method. Then it checks its *firstPageTemplate* property to find the first page to render. It renders the page by rendering all the components and StreamFrame objects assigned to it, in the order they were originally created (the same order as they appear in the class definition in the JRP file).

To render a StreamFrame object, IntraBuilder looks to its *streamSource* property. The Band objects in that StreamSource object—the *detailBand* and the *headerBand* and *footerBand* of any groups—are rendered in the StreamFrame object to fill it with data.

Before each component in the band is rendered, its *canRender* event fires.
The *canRender* event can be used to supplement the *suppressIfBlank* and *suppressIfDuplicate* properties of the HTML component by returning false, but it is more often used to alter the properties of a component just before it is rendered. For example, you can set a component's *color* to red if it's going to display a negative number. When used this way, the *canRender* event handler does what it wants and returns *true*, so that component is rendered. After the component is rendered, its *onRender* event fires. You can use the *onRender* event to reset the component to its original state.

Until the data from the StreamSource object is exhausted, that is unless the StreamSource object's *rowset* reaches the end-of-set, IntraBuilder knows that it needs to fill another StreamFrame. If there is another StreamFrame object in the same PageTemplate that used the same *streamSource*, the report engine will continue to stream bands from that StreamSource into that StreamFrame.

For example, if a PageTemplate has three tall StreamFrame objects side-by-side that have the same *streamSource* property, data would be printed in three columns on each page. To create a page of labels, create one StreamFrame for each label, all with the same *streamSource* property. Then set the *beginNewFrame* property of the *streamSource*'s *detailBand* to *true*, so that each row of data is rendered in a new StreamFrame.

If there are no more StreamFrame objects that can be filled on the current page, another page is scheduled. The current PageTemplate object's *nextPageTemplate* property refers to the PageTemplate to use.

Once the current page has finished rendering, the Report object's *onPage* event fires. If there is another page scheduled, it is rendered. Its StreamFrame objects are filled with data and the process repeats itself until all the StreamSource objects are exhausted. The *onPage* event fires one last time and the report is done.

class Band

Related topics

Contains the objects to output for a single row in a stream, or the header or footer of a group.

Syntax

These objects are automatically created by the StreamSource and Group objects.

Properties

The following table lists the properties of the Band class. (No events or methods are associated with this class.)

Property	Default	Description
beginNewFrame	false	Whether rendering always starts in a new StreamFrame
className	Band	Identifies the object as an instance of the Band class
expandable	true	Whether the band will increase in size automatically to accommodate the objects within it
height	0	The height of the band in twips
parent		The StreamSource or Group object that contains the Band
visible		Whether the band is visible

Description

Event	Parameters	Description

none

Method	Parameters	Description

none

A Band object acts as a container for visual components. They are created automatically for StreamSource and Group objects and cannot be created manually. There are three kinds of Band objects: detail bands, header bands, and footer bands.

A detail band is assigned to a StreamSource's *detailBand* property. The contents of the band are output once for each row in the StreamSource's rowset. Header and footer bands are assigned to a Group object's *headerBand* and *footerBand* properties respectively. They are rendered at the beginning and end of each group.

For a detail band, setting its *beginNewFrame* property to *true* causes each row from the StreamSource's rowset to be rendered in a new StreamFrame, which is the desired behavior when creating labels.

For a summary-only report, leave the detail band empty and set its *height* to zero.

Even if a band's *height* is set to zero, if its *expandable* property is true and it contains components, the band will expand to show those components.

class Band example

class Group

Related topics

Describes a group in a report.

Syntax

[<oRef> =] new Group(<streamSource>)

<oRef>

A variable or property—typically of <streamSource>—in which you want to store a reference to the newly created Group object.

<streamSource>

The StreamSource object to which the Group object binds itself.

Properties

The following tables list the properties and methods of the Group class. (No events are associated with this class.)

Property	Default	Description
className	Group	Identifies the object as an instance of the Group class.
footerBand		Specifies a Band that renders after a group of detail bands.
groupBy		A character string containing the field name by which groups are formed. If blank, the group is for the entire report.
headerBand		Specifies a Band that renders before a group of detail bands.
headerEveryFrame	false	Specifies whether to repeat the *headerBand* when a Group spans more than one StreamFrame.
name		The name of the Group object.
parent		The Report or StreamSource object that contains the Group.

Method	Parameters	Description
agAverage()	<codeblock>	Aggregate method that returns the mean average for a group
agCount()	<codeblock>	Aggregate method that returns the number of items in a group
agMax()	<codeblock>	Aggregate method that returns the highest value within a group
agMin()	<codeblock>	Aggregate method that returns the lowest value in a group
agStandardDeviation()	<codeblock>	Aggregate method that returns the standard deviation of the values in a group
agSum()	<codeblock>	Aggregate method that returns the total of a group
agVariance()	<codeblock>	Aggregate method that returns the variance of the values in a group
release()		Explicitly releases the Group object from memory

Description

Use Group objects to group data and calculate aggregate values for the group. Groups may be nested, and are handled in the order that they are created (the same order that they appear in the class definition in a JRP file).

The *groupBy* property contains the name of the field that defines the group, and may include an optional ascending or descending modifier. Whenever the value of that field changes, a new group starts. Therefore, the data must be sorted on the grouping field(s).

A Group object's *headerBand* is rendered before each group and its *footerBand* is rendered afterward. If the *headerEveryFrame* property is *true*, the group's *headerBand* is rendered at the beginning of every

StreamFrame.

If the Report object's *autoSort* property is true, data in a report is automatically sorted to match groups.

The Report object has its own Group object that is referred to by its *reportGroup* property. Its *groupBy* property is an empty sting, and the group is used for report-wide aggregates.

class Group example

class PageTemplate

Related topics

Describes the layout of a page of a report.

Syntax

[<oRef> =] new PageTemplate(<report>)

<oRef>

A variable or property—typically of <report>—in which you want to store a reference to the newly created PageTemplate object.

<report>

The Report object to which the PageTemplate object binds itself.

Properties

The following tables list the properties and methods of the PageTemplate class. (No events are associated with this class.)

Property	Default	Description
background		Background image when the report is rendered in HTML
className	PageTemplate	Identifies the object as an instance of the PageTemplate class
color	white	Background color for the page
gridLineWidth	1	Width of HTML table grid lines when report is displayed on the browser (0=no grid lines)
height		Height of the page in twips (1/20th of a point; 1440 twips/inch)
marginBottom	.75 inch = 1080 twips	The space between the bottom of the page and the usable area of the PageTemplate
marginLeft	.75 inch = 1080 twips	The space between the left side of the page and the usable area of the PageTemplate
marginRight	.75 inch = 1080 twips	The space between the right side of the page and the usable area of the PageTemplate
marginTop	.75 inch = 1080 twips	The space between the top of the page and the usable area of the PageTemplate
name		The name of the PageTemplate object
nextPageTemplate		The PageTemplate object that is used for the following page
parent		The Report object that contains the PageTemplate
width		Width of the page in twips (1/20th of a point; 1440 twips/inch)

Method	Parameters	Description
release()		Explicitly releases the PageTemplate object from memory

Description

A PageTemplate object describes the layout of a page, including its background color or image. It acts as a container for StreamFrame objects and visual components, which represent fixed output, such as a report date and page number.

The location of these objects is relative to (and restricted by) the four margin- properties that dictate the usable area of the page. Changing the *marginLeft* or *marginTop* will move everything that's inside the PageTemplate.

Although you may create multiple PageTemplate objects in a report, for example a different first page or alternating odd and even pages, the Report Designer currently does not support multiple PageTemplate objects visually.

class PageTemplate example

class Report

Related topics

A container and controller of report elements.

Syntax

[<oRef> =] new Report()

<oRef>

A variable or property in which you want to store a reference to the newly created Report object.

Properties

The following tables list the properties, events, and methods of the Report class.

Property	Default	Description
autoSort	true	Whether to automatically sort data to match specified groups
className	Report	Identifies the object as an instance of the Report class
endPage	–1	Last page number to render (–1 for no limit)
firstPageTemplate		Reference to the first PageTemplate object, which describes the first page
form		Reference to itself, to simplify generic object referencing
linkText	Next Page	HTML that is displayed in the link that is automatically generated when there is another page to view
output	Default	Target media (0=Window, 1=Printer, 2=Printer file, 3=Default, 4=HTML, 5=HTML file)
outputFilename		Name of file if output goes to printer or HTML file
printer		An object describing various printer output options
reportGroup		Reference to a Group object for the report as a whole, for master counts and totals
reportPage		Current page number being rendered
startPage	1	First page number to output
title		Title of the report; appears in the title bar of the preview window or browser

Event	Parameters	Description
onDesignLoad		After the report is first loaded into the Report Designer
onPage		After a page is rendered
preRender		Just before the report is rendered

Method	Parameters	Description
isLastPage()		Determines whether there are any more pages to render
release()		Explicitly releases the Report object from memory
render()		Generates the report

Description

A Report object acts as the controlling container for all the objects that make up the report, including data access, page layout, and data stream objects.

The *reportGroup* property refers to a report-level Group object that can be used for report-wide summaries. This Group object is created automatically.

To generate the report, call its *render*() method.

You can control the pages that are output by setting the *startPage* and *endPage* properties. If two parameters are passed to the JRP file, the standard bootstrap code generated for a JRP file will use those parameters as the *startPage* and *endPage* properties before calling the report's *render*() method.

class Report example

class StreamFrame

Related topics

Describes an area on a page into which output is streamed.

Syntax

[<oRef> =] new StreamFrame(<pageTemplate>)

<oRef>

A variable or property—typically of <pageTemplate>—in which you want to store a reference to the newly created StreamFrame object.

<pageTemplate>

The PageTemplate object to which the StreamFrame object binds itself.

Properties

The following table lists the properties of the StreamFrame class. (No events or methods are associated with this class.)

Property	Default	Description
borderStyle	Default	The border around the StreamFrame object (0=Default, 1=Raised, 2=Lowered, 3=None, 4=Single, 5=Double, 6-Drop Shadow, 7=Client, 8=Modal, 9=Etched In, 10=Etched Out)
className	StreamFrame	Identifies the object as an instance of the StreamFrame class
form		Reference to the report that contains the StreamFrame object
height	0	Height of the StreamFrame object in twips
left	0	The location of the left edge of the StreamFrame object in twips, relative to the PageTemplate's *marginLeft*
marginHorizontal	0	Horizontal margin inside the StreamFrame
marginVertical	0	Vertical margin inside the StreamFrame
name		The name of the StreamFrame object
parent		The PageTemplate object that contains the StreamFrame
streamSource		Reference to a StreamSource object that contains objects to be rendered in the StreamFrame
top	0	The location of the top edge of the StreamFrame object in twips, relative to the PageTemplate's *marginTop*
width		Width of the StreamFrame object in twips

Description

A StreamFrame object describes a rectangular region inside the margins of a PageTemplate into which data from a StreamSource object is rendered.

Although you may create multiple StreamFrame objects in a PageTemplate, the Report Designer currently does not support multiple StreamFrame objects visually.

class StreamFrame example

class StreamSource

Related topics

Describes a data source for streaming.

Syntax

[<oRef> =] new StreamSource(<report>)

<oRef>

A variable or property—typically of <report>—in which you want to store a reference to the newly created StreamSource object.

<report>

The Report object to which the StreamSource object binds itself.

Properties

The following tables list the properties and methods of the StreamSource class. (No events are associated with this class.)

Property	Default	Description
className	StreamSource	Identifies the object as an instance of the StreamSource class
detailBand		A Band object that corresponds to the *rowset*
name		The name of the StreamSource object
parent		The Report object that contains the StreamSource
rowset		The Rowset object that drives the StreamSource

Method	Parameters	Description
release()		Explicitly releases the StreamSource object from memory

Description

A StreamSource object acts as the common ground between a rowset that contains data you want to display and a band that contains components to display that data.

Every StreamFrame is assigned a StreamSource. The same StreamSource object may be assigned to multiple StreamFrame objects. The data from a StreamSource is rendered in all the StreamFrame objects that are linked to it.

A StreamSource object may contain Group objects that group data to perform aggregate functions.

class StreamSource example

agAverage()

Related topics Example

Aggregate method that returns the mean average for a group.

Syntax

<oRef>.*agAverage*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value to average.

Property of

Group

Description

Use *agAverage*() to calculate the mean average of the value returned by <codeblock> in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the Group object's *parent* StreamSource object's *rowset*.

If <codeblock> returns a *null* value, it is not considered in the average.

You may call *agAverage*() at any time. If necessary, the report engine will look ahead to calculate the result.

agAverage() example

Suppose you're reporting test scores, grouped by age. You display the average in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agAverage*() method:

```
{||this.parent.parent.agAverage({||
this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agAverage*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agAverage*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

agCount()

Related topics Example

Aggregate method that returns the number of items in a group.

Syntax

<oRef>.*agCount*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value you want to count.

Property of

Group

Description

Use *agCount*() to count the number of items in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If <codeblock> returns a *null* value, that item is not counted, so that empty rows will be skipped. To count a row even if it is empty, have the <codeblock> return a constant non-*null* value, for example,

```
{||1}
```

You may call *agCount*() at any time. If necessary, the report engine will look ahead to calculate the result.

agCount() example

Suppose you're reporting test scores, grouped by age. You display the number of tests scored in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agCount*() method:

```
{||this.parent.parent.agCount({||
this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agCount*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agCount*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

agMax()

Related topics Example

Aggregate method that returns the highest value within a group.

Syntax

<oRef>.*agMax*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value you want to track.

Property of

Group

Description

Use *agMax*() to return the highest value returned by <codeblock> in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If <codeblock> returns a *null* value, it is ignored.

You may call *agMax*() at any time. If necessary, the report engine will look ahead to determine the result.

agMax() example

Suppose you're reporting test scores, grouped by age. You display the highest score in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agMax*() method:

```
{||this.parent.parent.agMax({||this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agMax*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agMax*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

agMin()

Related topics Example

Aggregate method that returns the lowest value within a group.

Syntax

<oRef>.*agMin*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value you want to track.

Property of

Group

Description

Use *agMin*() to return the lowest value returned by <codeblock> in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If <codeblock> returns a *null* value, it is ignored.

You may call *agMin*() at any time. If necessary, the report engine will look ahead to determine the result.

agMin() example

Suppose you're reporting test scores, grouped by age. You display the lowest score in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agMin*() method:

```
{||this.parent.parent.agMin({||this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agMin*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agMin*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

agStandardDeviation()

Related topics Example

Aggregate method that returns the standard deviation of the values in a group.

Syntax

<oRef>.*agAverage*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value you want to sample.

Property of

Group

Description

Use *agStandardDeviation*() to calculate the standard deviation of the value returned by <codeblock> in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If <codeblock> returns a *null* value, it is not considered in the sample.

You may call *agStandardDeviation*() at any time. If necessary, the report engine will look ahead to calculate the result.

agStandardDeviation() example

Suppose you're reporting test scores, grouped by age. You display the standard deviation in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agStandardDeviation*() method:

```
{||this.parent.parent.agStandardDeviation({||
this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agStandardDeviation*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agStandardDeviation*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

agSum()

Related topics Example

Aggregate method that returns the total of a group.

Syntax

<oRef>.*agSum*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value you want to total.

Property of

Group

Description

Use *agSum*() to calculate the total of the value returned by <codeblock> in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If <codeblock> returns a *null* value, it is ignored.

You may call *agSum*() at any time. If necessary, the report engine will look ahead to calculate the result.

agSum() example

Suppose you're tracking overtime hours, grouped by employee. You display the average in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agSum*() method:

```
{||this.parent.parent.agSum({||
this.parent.rowset.fields[ "Overtime" ].value})}
```

To get to the Group object's *agSum*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agSum*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

agVariance()

Related topics Example

Aggregate method that returns the variance of the values in a group.

Syntax

<oRef>.*agVariance*(<codeblock>)

<oRef>

The Group object that defines the group you want to summarize.

<codeblock>

A codeblock or pointer to a function that returns the value you want to sample.

Property of

Group

Description

Use *agVariance*() to calculate the variance of the value returned by <codeblock> in the group. <codeblock> is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If <codeblock> returns a *null* value, it is not considered in the sample.

You may call *agVariance*() at any time. If necessary, the report engine will look ahead to calculate the result.

agVariance() example

Suppose you're reporting test scores, grouped by age. You display the variance in an HTML component in the group's *footerBand*. The *text* of the HTML component is an expression codeblock that calls the *agVariance*() method:

```
{||this.parent.parent.agVariance({||
this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agVariance*() method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agVariance*() also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

autoSort

Related topics

Whether to automatically sort data to match specified groups.

Property of

Report

Description

For groups to work properly, data must be sorted to match the groups.

If a Report object's *autoSort* property is *true* (the default), then the *sql* property of any query that is accessed by a StreamSource object that has groups will be modified automatically to include an ORDER BY clause that sorts the rowset in the correct order.

For example, if you have two Group objects, the first grouping by the field State and the second by Zip, then even if the query's *sql* property is set as:

```
select * from SALES
```

the rowset will actually be generated internally with the SQL statement:

```
select * from SALES order by STATE, ZIP
```

If *autoSort* is *false*, the rowset is not altered by the report engine. It assumes that the query is correct and contains the necessary fields in the right order. Therefore, if you use the *indexName* property to set the rowset order, you should set *autoSort* to *false*; otherwise it defeats the purpose of using *indexName*.

beginNewFrame

Related topics

Specifies whether rendering always starts in a new StreamFrame.

Property of

Band

Description

Set the *beginNewFrame* property of the StreamSource object's *detailBand* to *true* if you want each row to be rendered in its own StreamFrame.

If you have one StreamFrame in each PageTemplate, you will get one row per page. If you have multiple StreamFrames in each PageTemplate, each one will have at most one row of data.

You would create a page of labels by creating a StreamFrame for each label, set all the StreamFrame objects' *streamSource* property to the same StreamSource, and set its *detailBand*'s *beginNewFrame* property to *true*.

Set the *beginNewFrame* property of a group's *headerBand* to *true* if you want each group to start in a new StreamFrame. If you have one StreamFrame per page, that makes each group start on a new page.

If the *beginNewFrame* property of a group's *footerBand* is *true*, then whenever it is rendered, it will start in a new StreamFrame. For example, you could print a summary page for a report by creating a large *footerBand* for the Report object's *reportGroup* and set its *beginNewFrame* property to *true*.

borderStyle

Determines the border around the object.

Property of

HTML, StreamFrame

Description

borderStyle determines the display style of an object's rectangular frame. Set it to one of the following:

Value	Alignment
0	Default
1	Raised
2	Lowered
3	None
4	Single
5	Double
6	Drop shadow
7	Client
8	Modal
9	Etched in
10	Etched out

borderStyle has no effect when the object is rendered as HTML.

canRender

Related topics Example

Just before the component is rendered; return value determines whether the component is displayed.

Parameters

none

Property of

CheckBox, HTML, Image, Radio, Rule

Description

canRender fires for visual components only when they are in a report. It is fired every time the object is rendered. For a component in a detail band, that means for every row in the rowset.

While you can use *canRender* to evaluate some condition and return *false* to prevent the component from being displayed, the more common use of *canRender* is to alter a component's properties conditionally and always return *true*. You can create a calculated field in a report by altering an HTML component's *text* property in its *canRender* event handler.

You can use the *onRender* event to reset the component to its default state afterward, or always choose the desired state in the *canRender* event.

canRender example

Suppose you're printing a balance sheet and you want to highlight all the negative numbers by making them red. The default *color* property of the HTML component is black. Use the following *canRender* event handler to set the *color* property appropriately just before the component is rendered:

```
function someFigure_canRender()
{
  this.color = this.text() < 0 ? "red" : "black";
  return true;
}
```

Because the *text* property of the HTML component is an expression codeblock, to get the value that is going to be displayed, call the component's *text* property. Don't forget to *return true*; otherwise the component is never displayed.

detailBand

Related topics

The Band object in a StreamSource, which displays data from the rowset.

Property of

StreamSource

Description

A StreamSource object automatically has a Band object assigned to its *detailBand* property. This is the band that is rendered to display data in the rowset.

Visual components for displaying detail rows in the report should be created as a property of the StreamSource object's *detailBand*.

endPage

Related topics Example

The last page number to render.

Property of

Report

Description

By default, *endPage* is –1, which means that all the pages in the report are rendered. Set *endPage* to a number greater than zero to set the last page to render. When and if the report engine gets to that page, it stops after it has finished rendering it.

If the report is being rendered as HTML and it stops because it has reached its *endPage*, an HTML link will automatically be generated to print the next page if the report's *linkText* property is set.

When you run a JRP file with two parameters, the standard report bootstrap code assigns the second parameter to the report's *endPage* property just before it calls its *render*() method.

endPage example

The following statement runs a report, displaying only pages 6–10:

```
_sys.forms.run( "BIGLIST", 6, 10 );
```

expandable

Related topics

Specifies whether an object will increase in size automatically to accommodate the objects within it.

Property of

Band

Description

If a Band object's *expandable* property is true (the default), it will increase in size to display all the components inside it, even if its *height* is set to zero.

Set *expandable* to false if you want to make the number of rows displayed on each page constant, no matter what is displayed.

firstPageTemplate

Related topics

The PageTemplate object that is used for the report's first page.

Property of

Report

Description

Because a report may have multiple PageTemplate objects, the *firstPageTemplate* property is used to identify the PageTemplate that the report should render as its first page.

Once the first PageTemplate has been chosen, each PageTemplate object has a *nextPageTemplate* property that identifies the page to render next.

fixed

Related topics

Specifies whether an object's position in a band is fixed or if it can be "pushed down" or "pulled up" by the rendering or suppression other objects.

Property of

HTML

Description

Consider two components in a band named object1 and object2. Suppose that

- The bottom of object1 is at or above the *top* of object2.
- object1's *variableHeight* property is *true*.
- object1 grows in height to accommodate the data in it.
- The bottom of object1 is now below the *top* of object2.

Then if object2's *fixed* property is *false* (the default), object2 will be pushed down by object1 so that object2's *top* will be at the bottom of object1. This in turn might push down other objects in the band.

object2 can also be pulled up if the bottom of object1 is at or above the top of object2 and object1 is suppressed by its *suppressIfBlank* or *suppressIfDuplicate* properties.

The horizontal position of the objects in question doesn't matter, only the vertical position of their top and bottom ends.

If an object's *fixed* property is *true*, it is not moved by the rendering or suppression of other objects.

After the band has been rendered, all the objects return to their original positions and sizes.

footerBand

Related topics Example

The Band object that renders after each group.

Property of

Group

Description

A Group object automatically has a Band object assigned to its *footerBand* property. This band is rendered after each group is completed; that is, just before the next group starts or at the end of the rowset. It usually contains components that display summary information.

For components in the *footerBand*, the Group object's aggregate functions will return summary values for the group that has just completed.

footerBand example

Suppose you're printing a list of students grouped by grade. At the end of each grade, you want to display the number of students in that grade. In the Group object's *footerBand*, you create an HTML component with the following expression codeblock assigned to its *text* property:

```
{||"Count: " + this.parent.parent.agCount({||
this.parent.rowset.fields["ID"].value})}
```

groupBy

Related topics Example

The name of the field upon which groups are formed.

Property of

Group

Description

IntraBuilder groups rows by monitoring the value of a field in the rowset. The *groupBy* property contains the name of a field as a character string with an optional ascending or descending sort designation (not case-sensitive, ascending is the default). You may abbreviate ascending as “ASC” and descending as “DESC”. For example, if you’re grouping by the Age field, you could have one of the following strings as the *groupBy* property:

```
Age
Age asc
Age desc
Age ascending
Age descending
```

The ascending and descending options have an effect only if the Report’s *autoSort* property is set to *true*. In that case, IntraBuilder will make sure that the data in the rowset is sorted by the correct field in the correct direction.

No matter what the value of *autoSort* is, the named field must exist in the rowset. IntraBuilder looks for that field in the rowset’s *fields* array, just as you would. For example,

```
rowset.fields[ "Age" ].value
```

Because IntraBuilder uses the rowset’s *fields* array, you can group on calculated fields. There are two ways to do this. You can create a calculated field in a SQL SELECT statement, in which case set *autoSort* to *true*; or you can create the calculated field by adding a CalcField object to the rowset’s *fields* array, in which case you must set *autoSort* to *false*, because the named field doesn’t exist in the table, so you can’t sort on it. You would still have to make sure that the rows are sorted in the correct order so that all the rows in the same group are contiguous in the rowset.

groupBy example

Suppose you're tracking sales and want to generate a summary report, grouped by quarter. The data stores the actual date, so you'll need to calculate the quarter.

To calculate the quarter, divide the month of the date by 3. Because JavaScript months are zero-based, you then round that number down:

Month	Month number	Divided by 3	Rounded down
January	0	0	0
March	2	2/3	0
April	3	1	1
December	11	3 2/3	3

This returns a zero-based quarter number. The fact that the first quarter is numbered zero doesn't necessarily matter. To handle groups, the important factor is when the grouping value changes. You can add the calculated field in the query's *onOpen* event:

```
function sales1_onOpen()
{
  var c = new CalcField( "Quarter" );  // Create calc'd field
  this.rowset.fields.add( c );         // Add it to the fields array
  c.beforeGetValue = {||Math.floor( this.parent.fields[ "Date" ].value /
3 )};
}
```

The group's *groupBy* property is set to "Quarter". You still need to sort the report by date so that the groups will be in the right order. You can't use *autoSort*, since it will try to sort by a field named "Quarter", and there isn't one. So you use the following SQL SELECT statement in the query's *sql* property:

```
select * from OVERTIME order by OVERTIME."DATE"
```

DATE is an SQL reserved word, so you need to place the field name in quotes and use the table name.

headerBand

Related topics Example

The Band object that renders before each group.

Property of

Group

Description

A Group object automatically has a Band object assigned to its *headerBand* property. This band is rendered before the start of a new group, including the first group in the report; and if the Group object's *headerEveryFrame* property is true, at the beginning of every StreamFrame. It usually contains components that identify the current group or display summary information.

headerBand example

Suppose you're tracking sales and want to generate a summary report, grouped by quarter. You've already created a calculated field "Quarter" that contains a number from 0 to 3. To print "1st quarter", "2nd quarter" and so forth, set the *text* property of an HTML component to the following expression codeblock (all in one line):

```
{||{"1st","2nd","3rd","4th"}
[this.parent.parent.parent.rowset.fields["Quarter"].value] +
" quarter"}
```

This codeblock uses a literal array that contains the corresponding text for the zero-based quarter number. To get the quarter number in the calculated field "Quarter" from the HTML component:

- *this* is the component
- the component's *parent* is the *headerBand*
- the *headerBand*'s *parent* is the Group object
- the group's *parent* is the StreamSource object
- from the StreamSource object, you can access its *rowset*

headerEveryFrame

Related topics

Specifies whether to repeat a group's *headerBand* when a group spans more than one StreamFrame.

Property of

Group

Description

A group's *headerBand* is rendered at the beginning of the group. By default, *headerEveryFrame* is false; that means that if the contents of the group go into another frame, the *headerBand* is not repeated.

Set *headerEveryFrame* to *true* if you want a group's *headerBand* to be rendered at the top of every StreamFrame. For example, if you have one StreamFrame per page, setting *headerEveryFrame* to *true* will print the *headerBand* at the top of each page, underneath the fixed components (for example column headings) on the page.

If you have nested groups with *headerEveryFrame* set to *true* for each *headerBand*, the header bands will appear in group order at the top of every StreamFrame.

The *headerBand*'s *beginNewFrame* property determines whether the header band for a new group should start in a new StreamFrame. In contrast, *headerEveryFrame* determines whether the header band should be repeated in subsequent StreamFrame objects.

isLastPage()

Related topics Example

Returns *true* or *false* to let you know if additional pages are due to be rendered.

Syntax

<oRef>.*isLastPage*()

<oRef>

An object reference to the report you want to check.

Property of

Report

Description

isLastPage() returns *true* if the current page is the last page of the report, and *false* if additional pages are to be rendered.

Its main purpose is to allow you to make informed decisions about whether or not to display a custom link to additional pages. Use the report's *linkText* property for the default page link. You may also use *isLastPage*() to display something on the last page of a report.

IntraBuilder does not determine in advance how many pages a report will take. It renders the report one page at a time by filling all the StreamFrame objects on that page with data drawn from the StreamFrame objects' *streamSource*. If there is more data to render and all the StreamFrame objects in the page are full, another page is scheduled.

If the page being rendered is before the report's *startPage*, the rendering is not output. After the entire page has finished rendering, if another page is scheduled, it is rendered. The process repeats until all the pages are rendered, or the report's *endPage* page is rendered. In that case, the rendering process stops, even though another page may be scheduled.

isLastPage() ignores the *endPage* setting and determines if another page is scheduled to be rendered. It can be reliably called only after the last StreamFrame on a PageTemplate has been rendered, since it is the rendering of StreamFrame objects that determines the scheduling of new pages.

isLastPage() is usually called from the *canRender* event handler for an HTML component attached to the PageTemplate—not in a band—that is defined after all the StreamFrame objects. The order in which objects are created and assigned in the report class constructor directly determines their order of definition and rendering.

If you're creating your own link to the next page of the report, you should disable the automatic link by setting the report's *linkText* property to an empty string. Otherwise, both your link and the automatic link will appear.

isLastPage() example

Suppose you're displaying results from a query, 10 matches to a page. To display the next page of matches from a link, you pass the desired page number and the query string to the JRP. The query string has been converted into a URL-friendly format with the *escape()* function and stored as a property of the report in the report's Header.

An HTML component is placed in the PageTemplate, below the StreamFrame. The following method is the *canRender* event handler for that component.

```
function nextPageLink_canRender()
{
  if ( this.form.isLastPage() ) {          // If no more pages
    return false;                          // suppress link
  }
  else {
    var nPage = this.form.reportPage + 1;  // Next page to display
    this.text = '<A HREF="intrasrv.isv?knowbase/search.jrp(' +
                nPage + ',' + this.form.queryString +
                ')">Next 10 matches</A>';  // Create link
    return true;                           // OK to display link
  }
}
```

The HREF calls the report via the IntraBuilder Broker INTRASRV.ISV, which is valid for all protocols. The JRP is called in the directory where it's located, with the next page number and query string as parameters.

leading

Related topics

The distance between consecutive lines inside a component.

Property of

HTML

Description

leading controls the line spacing within an HTML component. By default, it's zero, which uses the default line space for the font.

You can set *leading* to a non-zero value to set the baseline-to-baseline distance of the text in the component. The *leading* property is in twips (20th of a point). There are exactly 1440 twips per inch.

leading has no effect when the object is rendered in HTML.

linkText

Related topics Example

HTML that is displayed in the link that is automatically generated when there is another page to view.

Property of

Report

Description

The *linkText* property contains a string that is used when

- A report is being rendered into HTML for viewing in a browser and
- It is forced to stop because it has reached the page designated by the *endPage* property and
- There are more pages to view

If *linkText* is not empty, then an HTML link is automatically generated that will render the next page of the report. It runs the same JRP with the next page number as the start and end page parameters. The contents of the *linkText* property is displayed as the link (that is, inside the <A> tags).

If *linkText* is empty (set to an empty string), no link is generated.

linkText may contain any string, including HTML to display an image.

linkText example

Suppose you have a suitable image to inform users that there is another page to view. You can use it in the *linkText* property, including alternate text if they have their graphics turned off:

```
<IMG SRC="NEXTPAGE.GIF" ALT="Next page">
```

marginBottom

Related topics

The space between the bottom of the page and the usable area of the PageTemplate.

Property of

PageTemplate

Description

Use *marginBottom* in conjunction with the PageTemplate's other margin- properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginBottom indicates the distance, in twips, between the bottom of the page and the bottom of the usable area. There are exactly 1440 twips per inch.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

The position of a page's usable area is irrelevant when the report is rendered as HTML; however the size of the usable area still dictates how much information will fit on each page.

marginHorizontal

Related topics

The horizontal margin between the edge of the object and its contents.

Property of

HTML, StreamFrame

Description

An object's horizontal margin is the same on both the left and right sides. In an HTML component, the text is indented inside its rectangular frame. The *left* position of all bands inside a StreamFrame object are relative to the horizontal margin on the left and restricted by the horizontal margin on the right.

marginHorizontal is measured in twips. There are exactly 1440 twips per inch.

The exact position of contents inside an object is irrelevant when the report is rendered as HTML; however the size of the margins still dictates how much information will fit in the object, and therefore on each page.

marginLeft

Related topics

The space between the left edge of the page and the usable area of the PageTemplate.

Property of

PageTemplate

Description

Use *marginLeft* in conjunction with the PageTemplate's other margin- properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginLeft indicates the distance, in twips, between the left edge of the page and the left edge of the usable area. There are exactly 1440 twips per inch.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

The position of a page's usable area is irrelevant when the report is rendered as HTML; however the size of the usable area still dictates how much information will fit on each page.

marginRight

Related topics

The space between the right edge of the page and the usable area of the PageTemplate.

Property of

PageTemplate

Description

Use *marginRight* in conjunction with the PageTemplate's other margin- properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginRight indicates the distance, in twips, between the right edge of the page and the right edge of the usable area. There are exactly 1440 twips per inch.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

The position of a page's usable area is irrelevant when the report is rendered as HTML; however the size of the usable area still dictates how much information will fit on each page.

marginTop

Related topics

The space between the top of the page and the usable area of the PageTemplate.

Property of

PageTemplate

Description

Use *marginTop* in conjunction with the PageTemplate's other margin- properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginTop indicates the distance, in twips, between the top of the page and the top of the usable area. There are exactly 1440 twips per inch.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

The position of a page's usable area is irrelevant when the report is rendered as HTML; however the size of the usable area still dictates how much information will fit on each page.

marginVertical

Related topics

The vertical margin between the edge of the object and its contents.

Property of

HTML, StreamFrame

Description

An object's vertical margin is the same on both the top and bottom. All rendering in the object, whether it's text in an HTML component or bands inside a StreamFrame object, is relative to the vertical margin on the top and restricted by the vertical margin on the bottom.

marginVertical is measured in twips. There are exactly 1440 twips per inch.

The exact position of contents inside an object is irrelevant when the report is rendered as HTML; however the size of the margins still dictates how much information will fit in the object, and therefore on each page.

nextPageTemplate

Related topics

The PageTemplate object that is used for the following page.

Property of

PageTemplate

Description

Because a report may have multiple PageTemplate objects, the *firstPageTemplate* property is used to identify the PageTemplate that the report should render as its first page.

Once the first PageTemplate has been chosen, each PageTemplate object has a *nextPageTemplate* property that identifies the page to render next.

For a report that uses the same page over and over, the same PageTemplate object is used for both the report's *firstPageTemplate* property and that PageTemplate's own *nextPageTemplate* property.

You can create a different introduction or cover page for a report by specifying the cover page as the report's *firstPageTemplate* property, and then set the cover page's *nextPageTemplate* property to the PageTemplate for the body pages.

To alternate left and right pages, set the *nextPageTemplate* of the left page to the right page, and vice versa. Then specify the page on the right as the report's *firstPageTemplate*.

onPage

Related topics Example

After the page has finished rendering.

Parameters

none

Property of

Report

Description

onPage fires after each page has finished rendering, including the last page. By that time, it's too late to do anything to the page, but you can take actions for the next page.

With the *preRender* event, which fires once at the beginning of the report, you have the opportunity to take actions before and after every page in the report.

In an *onPage* event handler, the report's *reportPage* property indicates the page that has just finished rendering.

onPage example

Suppose you're going to print a report, make two-sided copies, and bind them. You want to shift the margins slightly on both pages to accommodate the binding. The right pages need to move to the right and the left pages need to move to the left. Other than that, the pages are identical.

You can use the *onPage* event handler to shift the margins of the PageTemplate after each page has printed, in preparation for the next page:

```
function Report_onPage()
{
  #define TWIPS(x) ((x)*1440)
  if ( this.reportPage % 2 == 0 ) {  // Finished left page, start right
    this.pageTemplate1.marginLeft  = TWIPS( 1.0 );
    this.pageTemplate1.marginRight = TWIPS( 0.5 );
  }
  else {                             // Finished right page, start left
    this.pageTemplate1.marginLeft  = TWIPS( 0.5 );
    this.pageTemplate1.marginRight = TWIPS( 1.0 );
  }
}
```

Pages on the left are even-numbered. An even number modulo 2 yields zero, so if a left page has just been printed, the margins are set to those for a right page, and vice versa.

To set the page margins for the first page, call the *onPage* event handler from the report's *preRender* event:

```
function Report_preRender()
{
  this.onPage();  // Call onPage event handler
}
```

In the *preRender* event, the page number is zero, so that sets the first page to print on the right, as it should.

onRender

Related topics Example

After the component is rendered.

Parameters

none

Property of

CheckBox, HTML, Image, Radio, Rule

Description

onRender fires for visual components only when they are in a report. It is fired every time the object is rendered, after it has finished rendering. For a component in a detail band, that means for every row in the rowset.

You can use the *onRender* event to reset the component to its default state after changing it in its *canRender* event.

onRender example

Suppose you're printing a list of test scores, grouped by age. You have a *headerBand* that prints in every StreamFrame. After printing in the first StreamFrame, you want it to add the word "continued". Every time a new group starts, you want to remove the word. There is also a *footerBand* to print totals for the group.

You create an HTML object with the following expression codeblock as its *text* property:

```
{||"Age: " + this.parent.parent.parent.rowset.fields["Age"].value}
```

In the component's *onRender* event, you change the codeblock to include the word "continued":

```
function header1_html1_onRender()
{
  this.text = {||"Age: " +
this.parent.parent.parent.rowset.fields["Age"].value + " continued"}
}
```

So now, once it is rendered at the beginning of the group, it is changed so that it will contain the word "continued" for the rest of the group.

To change it back for the start of a new group, use the following *onRender* event handler for the HTML component in the *footerBand*:

```
function footer1_html1_onRender()
{
  this.text = {||"Age: " +
this.parent.parent.parent.rowset.fields["Age"].value};
}
```

This restores the original codeblock at the end of the group, preparing the HTML component for the beginning of the next group.

output

Related topics

Designates the target medium for the report.

Property of

Report

Description

Set the report's *output* property to designate how you want the report to be rendered. *output* may contain one of the following values:

Value	Target
0	Window
1	Printer
2	Printer file
3	Default
4	HTML
5	HTML file

The default output for a report depends on where it is called. When a report is run in the IntraBuilder Designer, it defaults to a preview window. When a report is run by an IntraBuilder Agent, it defaults to HTML.

If you designate either Printer file or HTML file, the report's *outputFilename* property must be set to the name of the target file.

outputFilename

Related topics

The name of the target output file.

Property of

Report

Description

If a report's *output* property is set to either Printer file or HTML file, the *outputFilename* property must be set to the target file.

The file will contain the output from the report. If the file already exists, it will be overwritten.

printer

Related topics

An object that describes various printer output options.

Property of

Report

Description

A *printer* object contains properties for the following printer options:

Property	Default	Description
color	Monochrome	Whether the output should be in color/grayscale or plain monochrome (0=Default, 1=Monochrome, 2=Color)
copies	1	Number of copies
duplex	None	Whether to print in duplex, and in which orientation (0=Default, 1=None, 2=Vertical, 3=Horizontal)
orientation	Portrait	The orientation of the output (0=Default, 1=Portrait, 2=Landscape)
paperSize	printer-dependent	The size of the paper to use
paperSource	printer-dependent	The paper tray or bin to use
printerName		The name of the target printer (blank for default printer)
printerSource	IntraBuilder default	Which printerName to use (0=IntraBuilder default, 2=Windows default, 3=Specific)
resolution	High	Graphics resolution (0=Default, 1=Draft, 2=Low, 3=Medium, 4=High)
trueTypeFonts	Outline	How to handle TrueType fonts (0=Default, 1=Bitmap, 2=Download, 3=Substitute, 4=Outline)

render()

Related topics

Renders the reports to the designated target.

Syntax

<oRef>.*render*()

<oRef>

An object reference to the report you want to render.

Property of

Report

Description

Call a Report object's *render*() method to generate the report. The output of the report goes to the target designated by the report's *output* property.

The report's *preRender* event fires just before the report engine begins to process data.

The report engine renders the report internally and outputs the results starting with the page designated by the *startPage* property. It continues to render all the objects on each page until it exhausts all StreamSource objects, or it has finished rendering the page designated by the *endPage* property.

If the report is being rendered as HTML and it stops because it has reached its *endPage*, an HTML link will automatically be generated to print the next page if the report's *linkText* property is set.

The standard bootstrap code renders an instance of the report when you run the report's JRP script file, so to render a form, run the JRP with *_sys.reports.run*() or *_sys.scripts.run*(). If you pass two parameters when you run the JRP, the standard report bootstrap code assigns them to the report's *startPage* and *endPage* properties just before it calls its *render*() method.

reportGroup

A Group object for the report as a whole.

Property of

Report

Description

A Report object automatically has a Group object assigned to its *reportGroup* property. The *groupBy* property of this Group object is an empty string.

Use the *reportGroup* to calculate aggregates for the entire report, for example, a grand total; and for items in a report introduction or summary.

A *reportGroup* has a *headerBand*, a *footerBand*, and aggregate methods, just like any other Group object. Because the *parent* of the *reportGroup* is the Report object instead of a StreamSource or Group object, the object reference path to data is slightly different for components in the *reportGroup*.

reportPage

Example

The current page number being rendered.

Property of

Report

Description

The *reportPage* property contains the number of the page that is being rendered.

During a report's *preRender* event, the *reportPage* is zero. During an *onPage* event, the *reportPage* is the page that has just finished rendering.

reportPage example

To display the current page number on a report, create an HTML component on the PageTemplate that contains the following expression codeblock as its *text* property:

```
{||"Page " + this.parent.parent.reportPage}
```

rotate

The orientation of an object, in 90-degree increments.

Property of

HTML

Description

To rotate the text inside an HTML component, set its *rotate* property to one of the following values:

Value	Clockwise rotation
0	none
1	90 degrees
2	180 degrees
3	270 degrees

rotate has no effect when the object is rendered in HTML.

startPage

Related topics Example

The first page number to output.

Property of

Report

Description

By default, *startPage* is 1, which means that all the pages that are rendered are output. Set *startPage* to a number greater than 1 to delay the beginning of the output.

The report engine must still render each page until it gets to the *startPage* because it dynamically paginates the report. The position of a row in a report may change whenever someone changes the table, so use caution when using *startPage* to display segments of a report.

When you run a JRP file with two parameters, the standard report bootstrap code assigns the first parameter to the report's *startPage* property just before it calls its *render*() method.

startPage example

The following statement runs a report, displaying only pages 6 through 10:

```
_sys.forms.run( "BIGLIST", 6, 10 );
```

streamSource

The StreamSource object that contains objects to render in the StreamFrame.

Property of

StreamFrame

Description

A StreamFrame object's *streamSource* property identifies the StreamSource object that supplies the StreamFrame object's data stream.

If multiple StreamFrame objects have the same *streamSource* property, that StreamSource will stream data to those StreamFrame objects in series, in the order in which the StreamFrame objects were created (the same order as they are listed in the class definition in the JRP file).

If multiple StreamFrame objects have different *streamSource* properties, each StreamFrame will be filled from its own StreamSource object (in the same order as above) until all the StreamFrame objects in the page are filled. If a particular StreamFrame object's StreamSource is exhausted, it is no longer filled. When all StreamSource objects are exhausted, all the objects on that last PageTemplate are rendered, and the report is done.

suppressIfBlank

Related topics

Specifies whether an object is suppressed, or not rendered, if it is blank.

Parameters

none

Property of

HTML

Description

suppressIfBlank has an effect only for visual components that are in a report. If *true*, it suppresses the rendering of the component if its display value is blank.

For example, suppose you're printing two-line addresses with the city underneath. If the second address line is blank, you don't want it to occupy any space. By setting that component's *suppressIfBlank* property to *true*, if it's blank, nothing gets rendered and all the components below it that have their *fixed* properties set to *false* are moved up to fill the space.

By default *suppressIfBlank* is *false*. You can also use the component's *canRender* event to suppress rendering.

suppressIfDuplicate

Related topics

Specifies whether an object is suppressed, or not rendered, if its value is the same as the previous time it was rendered.

Parameters

none

Property of

HTML

Description

suppressIfDuplicate has an effect only for visual components that are in a report. If *true*, it suppresses the rendering of the component if its display value is the same as the previous time it was rendered in the same report, even if the previous time was in another group in the report.

Use *suppressIfDuplicate* to eliminate duplicating the same data value over and over again for multiple rows in a report. For example, you may have information sorted by date. With *suppressIfDuplicate* set to *true*, each date will be rendered only once.

By default *suppressIfDuplicate* is *false*. You can also use the component's *canRender* event to suppress rendering.

tracking

Related topics

The amount of extra space between characters.

Property of

HTML

Description

tracking adds extra space between characters within an HTML component. By default, it's zero, which means no extra spacing.

You can set *tracking* to a non-zero value to add extra space between characters. The *tracking* property is in twips (20th of a point). There are exactly 1440 twips per inch.

tracking has no effect when the object is rendered in HTML.

trackJustifyThreshold

Related topics

The maximum amount of added space allowed between words in a fully justified line. Exceeding that amount switches to character tracking.

Property of

HTML

Description

trackJustifyThreshold sets a threshold for the amount of extra space between words that can be added to try to justify the line. If a line requires more than the threshold amount, the line is justified by adding space between each character in the line, in addition to the maximum space between each word.

If a line contains only one word and *trackJustifyThreshold* is non-zero, the word will be fully justified with character tracking, unless it is on the last line of text. The last line of text is never justified.

An HTML component's *alignHorizontal* property must be set to Justify in order for *trackJustifyThreshold* to have any effect.

The *trackJustifyThreshold* property is in twips (20th of a point). There are exactly 1440 twips per inch.

trackJustifyThreshold has no effect when the object is rendered in HTML.

variableHeight

Related topics

Whether an object's *height* can increase automatically to accommodate its contents.

Property of

HTML

Description

Set *variableHeight* to *true* so that an object can grow to accommodate its contents. If an object's *height* is not large enough to display everything, it is increased. *variableHeight* does not shrink objects to fit their contents.

If the object is in a Band object in a report and it grows, it might push down other objects in the band if those objects have their *fixed* property set to *false*.

By default *variableHeight* is *false*.

variableHeight has no effect when the object is rendered in HTML, but HTML acts like *variableHeight* is *true*. It always displays all the data that is sent to it.

verticalJustifyLimit

Related topics

The maximum amount of added space allowed between lines in a vertically justified object. Exceeding that amount makes the text top-aligned instead.

Property of

HTML

Description

verticalJustifyLimit sets the maximum amount of extra space between lines that can be added to try to vertically justify the lines in an object. If the maximum amount does not justify the lines, IntraBuilder gives up and makes the text top-aligned instead.

An HTML component's *alignVertical* property must be set to Justify in order for *verticalJustifyLimit* to have any effect.

The *verticalJustifyLimit* property is in twips (20th of a point). There are exactly 1440 twips per inch.

verticalJustifyLimit has no effect when the object is rendered in HTML.

Server-side extensions

While Java applets and ActiveX controls enhance IntraBuilder's capabilities on client browsers, you can use OLE Automation and the *extern* system to extend IntraBuilder server-side.

OLE Automation and the *extern* system allow you to use OLE2 servers and external DLLs as native IntraBuilder objects and functions.

class OleAutoClient

Example

Creates an OLE2 controller that attaches to an OLE2 server.

Syntax

[<oRef> =] new OleAutoClient(<server expC>)

<oRef>

A variable or property in which you want to store a reference to the newly created OleAutoClient object.

<server expC>

The name of the OLE Automation server. The name is of the form, “app.object”; for example, “word.basic”

Properties

The properties, events, and methods of each instance of the OleAutoClient class depend on the attached OLE automation server.

Description

OLE automation allows you to control another application, an OLE automation server, through an OLE automation client. For example, with a full-featured word processor as an OLE automation server, you could do the following all on the server machine:

- Start the word processor
- Open an order form
- Fill in data that was entered in a client browser
- Fax that order form to a customer
- Close the word processor

With IntraBuilder as the host for the OLE automation client, you could control the entire process from a browser. You don’t even need the word processor to be installed on the IntraBuilder client, just on the IntraBuilder server machine.

IntraBuilder’s dynamic object model is a natural host for OLE automation clients. Because there is no need to declare the capabilities of the OLE automation server as you would with a statically linked language, you can specify any OLE Automation server at run time, and use whatever capabilities it has.

Once you create the OleAutoClient object, the properties, events, and methods the OLE automation server provides are accessed through the OleAutoClient object, just like with stock IntraBuilder objects. You can *inspect*() the OleAutoClient object’s properties.

class OleAutoClient example

Suppose the Human Resources (HR) department has a number of forms that they have created with Microsoft Word. Whenever an employee wants a copy of a form, they go to the HR page of the company intranet and choose the form and the network printer they want.

The page is an IntraBuilder form that drives a copy of Microsoft Word through OLE Automation. In building the list of HR forms (from a table maintained by HR) and printers (from a table maintained by Information Services), the IntraBuilder form (created by the HR manager) builds two corresponding associative arrays to quickly lookup the actual file and printer names. Once the choice has been made, the file is opened, the printer is selected, the file is printed, and the file is closed with the following code:

```
function printButton_onServerClick()
{
  var w = new OleAutoClient( "word.basic" );
  w.FileOpen( this.form.aDocs[ this.form.docSelect.value ] );
  w.FilePrintSetup( this.form.aPrinter[ this.form.printerSelect.value ] );
  w.FilePrint();
  w.FileClose();
}
```

extern

Example

Declares a prototype for a non-IntraBuilder function contained in a DLL file.

Syntax

extern [*cdecl* | *pascal* | *stdcall*] <return type> <function name>
([<parameter type> [, <parameter type> ...]])
<filename expC>

or

extern [*cdecl* | *pascal* | *stdcall*] <return type> <user-defined function name>
([<parameter type> [, <parameter type> ...]])
<filename expC>
from <export function name> | <ordinal number>

Because you create a function prototype with *extern*, parentheses are required as with other functions.

cdecl | *pascal* | *stdcall*

Sets the function calling convention. The default is *stdcall*.

<function name>

The export name of the function. The export name of an external function is contained in the DEF file associated with the DLL file that holds the function, or explicitly exported in the source code.

<return type> and <parameter type>

A keyword representing the data type of the value returned by the function, and the data type of each argument you send to the function, respectively. The types match those in C, and for most data types you can pass either the value or a pointer to the value. The following table lists the keywords you can use.

Keyword	as pointer	IntraBuilder data type	Data type size
Parameters or return values			
int	*int **	Numeric	4 bytes (32 bits)
long	*long **	Numeric	4 bytes (32 bits)
short	*short **	Numeric	2 bytes (16 bits)
char		String	1 byte (8 bits)
	*char **	String	Null-terminated
unsigned int	*unsigned int **	Numeric	4 bytes (32 bits)
unsigned long	*unsigned long **	Numeric	4 bytes (32 bits)
unsigned short	*unsigned short **	Numeric	2 bytes (16 bits)
unsigned char		String	1 byte (8 bits)
	*unsigned char **	String	Null-terminated
float	*float **	Numeric	4 bytes (32 bits)
double	*double **	Numeric	8 bytes (64 bits)
long double	*long double **	Numeric	10 bytes (80 bits)
boolean	*boolean **	Logical	1 byte (8 bits)
void		none	N/A
Parameters only			
	*void **	String	
...		N/A	

In most cases, if the function expects a pointer as a parameter, IntraBuilder will pass a pointer to the value. If a function returns a pointer, IntraBuilder will get the value at the pointer and convert it into the appropriate IntraBuilder data type.

To pass or return a string, use the *char ** type.

If the function expects a pointer to a structure or some other data type that is not listed above, declare the parameter as *void **.

If the function has no parameters or returns no value, declare the data type as *void*.

You may use the ... parameter declaration if the calling convention is *stdcall* to designate a variable number of parameters.

<filename expC>

A character string containing the name of the DLL file in which the external function is stored. If the extension is omitted, the default is DLL. The file name of any DLL that you load in memory must be unique; for example, you can't load SCRIPT.DLL and SCRIPT.FON into memory concurrently, even though they have different file-name extensions.

If the DLL file is not already loaded into memory, *extern* loads it automatically. If the DLL file is already in memory, *extern* increments the reference counter.

The reference counter is incremented only the first time, regardless of how many times you execute the *extern* statement.

You may include a path in <filename>. If you omit the path, IntraBuilder looks in the following directories for the DLL by default:

1 The current directory.

2 The Windows directory (for example, C:\WINDOWS).

3 The Windows SYSTEM subdirectory (for example, C:\WINDOWS\SYSTEM).

4 The directory containing INTRA.**EXE.**

5 The directories in the current DOS path.

6 The directories mapped for search in a network.

The path specification is necessary only when the DLL file is not in one of these directories.

<user-defined function name>

The name you give to the external function instead of the export name. When you specify <user-defined function name> (instead of <function name>), you must use the *from* <export function name> | <ordinal number> clause to identify the function in the DLL file.

from <export function name> | <ordinal number>

Identifies the function in the DLL file specified by <filename>. <export function name> identifies the function by its name, which is stored in the DEF file that is associated with the DLL file. <ordinal number> identifies the function with a number, which is also stored in the DEF file.

extern example

Use *extern* to declare a prototype for an external function written in a language other than JavaScript. A prototype tells IntraBuilder to convert its arguments to data types the external function can use, and to convert the value returned by the external function into a data type IntraBuilder can use.

To call an external DLL function, first prototype it with *extern*. Then, using the name of the function you specified with *extern*, call the function as you would any IntraBuilder function. You must prototype an external function before you can call that function in IntraBuilder.

The external function may be in any 32-bit DLL, such as the Windows API or a third-party DLL file. Although most library code is contained in files with extensions of DLL, such code can be held in EXE files, or even in DRV or FON files.

Example

The following statements declare two external functions from a library named REQSTURL.DLL. requestUrl() gets a document from a Web server, and isLocal() determines whether the host is local. Note the use of in-line comments to document what each parameter means

```
extern char* requestUrl( char* /*host*/, char* /*file name*/, unsigned int
/*port*/ )
  "reqstURL.dll";
extern boolean isLocal( char* /*host*/ ) "reqstURL.dll";
```

Once the functions have been *extern*ed, you may call them like any other JavaScript function. For example,

```
cHTML = requestURL( "www.borland.com", "/intrabuilder/index.html", 80 );
```

_sys

Example

The global object representing the currently running instance of the IntraBuilder Designer or IntraBuilder Agent.

Syntax

The *_sys* object is automatically created when you start IntraBuilder.

Properties

The following tables list the properties and methods of the *_sys* object. (No events are associated with the *_sys* object.

Property	Default	Description
className	APPLICATION	Identifies the object as an instance of the IntraBuilder application
databases		An array containing references to all database objects used by the IntraBuilder Explorer
env		An object containing various environment methods
forms		An object containing methods related to forms
images		An object containing various image methods
os		An object containing methods related to the operating system
printer		An object containing properties and methods related to the printer
protection		An object containing methods related to table security and encryption
queries		An object containing methods related to queries
reports		An object containing methods related to reports
scriptOut		An object representing the Results pane of the Script Pad
scripts		An object containing methods related to scripts
session		The Script Pad's Session object
tables		An object containing methods related to tables

Method	Parameters	Description
inspect()	<oRef>	Inspects the properties of the object

Description

The *_sys* object is a container for system- and environment-level properties and methods that provide access to IntraBuilder functionality. You can write and execute *_sys* object statements in the Script Pad and include them in scripts. User actions in the IntraBuilder Explorer, such as double-clicking on a form, cause the equivalent *_sys* statement to be streamed out to the Script Pad and executed.

_sys example

To run a form, call the *run*() method of the *_sys* object's *forms* object. The following statement runs the BIOLIFE.JFM form:

```
_sys.forms.run( "BIOLIFE" );
```

env.getEnv()

Related topics Example

Returns the value of a DOS environment variable.

Syntax

_sys.env.getEnv(<expC>)

<expC>

The name of the DOS environment variable you want to evaluate.

Description

Use *env.getEnv*() to return the current value of a DOS environment variable.

DOS environment variables contain strings which are used by the operating system and application programs. For example, you may have a PATH string that contains the application program search path used by the operating system, a TMP string which contains the name of a directory that can be used for temporary files, and a BLASTER string that contains the hardware settings for your sound controller. These variables are created during Windows start-up with DOS commands like PATH, PROMPT, and SET in the AUTOEXEC.BAT file, and by the operating system. Although you may start a DOS window and alter the environment strings in that DOS session, those changes have no effect on other DOS windows or other applications like IntraBuilder; each task gets its own copy of the environment variables after start-up.

Although environment variable names are case-sensitive, IntraBuilder will attempt to find a match regardless of case. If IntraBuilder can't find the environment variable specified by <expC>, it returns an empty string.

env.getEnv() example

Suppose an application creates a temporary file to process data. This code excerpt looks for the DOS environment variable TMP to determine if there is a directory designated for temporary files. If not, the file is created in the current directory.

```
// Look for DOS environment variable TMP
var cTempDir = new StringEx( _sys.env.getEnv( "TMP" ) );
// If defined, make sure it has trailing backslash
if ( cTempDir != "" ) {
  if ( cTempDir.right( 1 ) != "\\" ) {  // No trailing backslash
    cTempDir += "\\";                   //   so add one
  }
}
// Create work file in temp directory
var fTemp = new File();
fTemp.create( cTempDir + "DATA.TMP" );
// ...and use it...
```

env.home()

Example

Returns the home directory of the currently running instance of IntraBuilder.

Syntax

_sys.*env*.*home*()

Description

Use *env*.*home*() to identify the directory in which the currently running copy of IntraBuilder is located. When you install IntraBuilder, the installation program (by default) installs in the home directory (Program Files\Borland\IntraBuilder) on drive C, creating subdirectories such as \Bin and \Server in which the various files are stored. *env*.*home*() returns the drive and home directory:

```
C:\Program Files\Borland\IntraBuilder\
```

with the trailing backslash.

env.home() example

Suppose you're writing a source code document and you want to list the contents of files that have been included using the #include preprocessor directive. These files are usually in the current directory, or in the \Include subdirectory off the home IntraBuilder directory.

```
// var cIncludeFile contains file name
// listFile() is your own function that lists the contents of a file
var fInclude = new File();
if ( fInclude.exists( cIncludeFile ) ) {
  listFile( cIncludeFile );
}
else if ( fInclude.exists( _sys.env.home() + "INCLUDE\\" + cIncludeFile ) ) {
  listFile( _sys.env.home() + "INCLUDE\\" + cIncludeFile );
}
else {
  alert( "Include file " + cIncludeFile + " not found." );
}
```

env.id()

Example

Returns the name of the current user on a local area network (LAN) or other multiuser system.

Syntax

_sys.*env*.*id*()

Description

env.*id*() returns the name of the current user as a character string. *env*.*id*() returns an empty string when the IntraBuilder Designer or IntraBuilder Agent is running on a single-user system or when a user name is not registered on a multiuser system.

Because *env*.*id*() is a server-side method, it cannot determine the user name of the person using the client browser. *env*.*id*() will return the name of the user that started the instance of the IntraBuilder Agent that is servicing the client.

env.id() example

Executing the following statement in the Script Pad displays the current user name in the Results pane:

```
? _sys.env.id()
```

env.memory()

Example

Returns the amount of currently available memory.

Syntax

_sys.*env*.*memory*([<expN>])

<expN>

Any number, which causes *env*.*memory*() to return the amount of available physical memory.

Description

Use *env*.*memory*() to determine how much memory is available in the system. It returns the amount in kilobytes (1024 bytes).

In Windows, available memory is a combination of physical memory (RAM installed in the computer) and virtual memory (disk space used to simulate memory).

When called with no parameters, *env*.*memory*() returns the total amount of available memory: the amount of unused physical memory plus the amount of disk space available for virtual memory. By default, Windows 95 sets no maximum for virtual memory, so *env*.*memory*() will return free physical memory plus free disk space on the hard drive used for virtual memory. On Windows NT, the size of the paging file used for virtual memory is set to a reasonable size.

When called with any numeric parameter, *env*.*memory*() returns the amount of free physical memory. The amount of free physical memory can vary greatly, depending on what the system is doing or has just finished doing. For example, you may have more free physical memory right after viewing and dismissing a dialog box, since the memory that was used to display the dialog box is momentarily unallocated.

IntraBuilder's About dialog box displays the amount of free physical memory (in bytes) and percentage of free GDI and User resources. There is no built-in method that returns free resources, although you can use *extern* to call the appropriate Windows API function from IntraBuilder.

env.memory() example

With Windows 95, 300 MB of free disk space, and 32 MB of RAM, executing the following statements in the Script Pad would produce results close to the values shown:

```
? _sys.env.memory()   // displays 304000 = (300 MB free disk space + 4 MB
RAM) / 1024
? _sys.env.memory(1)  // displays   4000 = 4 MB RAM / 1024
```

env.os()

Related topics Example

Returns the name and version number of the current operating system.

Syntax

_sys.*env*.*os*()

Description

Use *env*.*os*() to determine the version of Windows in which the IntraBuilder Designer or IntraBuilder Agent is running. To determine which version of IntraBuilder is running, use *env*.*version*(). *env*.*os*() returns a character string like:

```
Windows NT version 4.00
```

with the name of the operating system, the word “version” and the version number.

env.os() example

Executing the following statement in the Script Pad displays the name and version of the operating system in the Results pane:

```
? _sys.env.os()
```

env.version()

Related topics Example

Returns the name and version number of the currently running copy of IntraBuilder.

Syntax

_sys.env.version([<expN>])

<expN>

Any number, which causes *env.version*() to return extended version information.

Description

Although you may be able to use *env.version*() in programs to take advantage of version-specific features, the most common use of *env.version*() is to get the exact build number of your copy of IntraBuilder to see if you have the latest build. When called with no parameters, *env.version*() returns a string like:

```
IntraBuilder 1.0
```

with the product name and the version number. If you pass a number, for example *env.version*(1), you will get extended build information, like:

```
IntraBuilder 1.0  b84 (06/06/96-US960606)
```

which adds the build number after the “b”, the date of the build, and the identifier and date of the language resource for that copy of IntraBuilder. If you pass the number .89, you will get the build information for the Borland Database Engine used, for example,

```
BDE version: 05/06/96
```

env.version() example

When asking technical support questions, it's helpful to know the exact build number of IntraBuilder you're using. The problem you're having may be a known problem, and may already be fixed in a later build. To get the build number, execute the following statement in the Script Pad:

```
? _sys.env.version(1)
```

and read the build number after the "b" in the Results pane.

forms.design()

Related topics Example

Opens the Form Designer to create or modify a form.

Syntax

_sys.forms.design([<filename expC> [,<custom expL>]])

<filename expC>

The form you want to create or modify. If you leave out this parameter or pass an empty string, a new untitled form is created. You may specify "?" or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JFM.

<custom expL>

If true, opens the Form Designer in custom form mode. For custom forms, IntraBuilder assumes a JCF extension. The default is false.

Description

Use *forms.design*() to open the Form Designer and create or modify a form interactively. The Form Designer automatically generates an IntraBuilder script that defines the contents and format of a form, and stores this code in an editable text file with a JFM extension.

When a file name is specified, the presence of that file determines whether a create or modify operation occurs. If the JFM file exists, it is modified in the Form Designer. If the file doesn't exist, it is created.

When opened in custom form mode, the Form Designer works the same way, but saves the form definition in a JCF file instead of a JFM file. The JCF file does not create a form when run directly; instead the form class in the JCF file is used as a base form class for other forms.

Because JFM and JCF files are scripts, you can edit them with *scripts.design*().

Note The Forms Designer is a two-way tool. You can open a form in the Form Designer even if you've edited the code in the JFM or JCF file.

forms.design() example

To open a new untitled form design surface, execute the following statement in the Script Pad:

```
_sys.forms.design()
```

forms.expert()

Related topics Example

Opens the Form Expert to automatically create a form.

Syntax

_sys.forms.expert([<filename expC> [,<prompt expL> [,<home page expL>]]])

<filename expC>

The name of the form you want to create. If you leave out this parameter or pass an empty string, a new untitled form is created. If you specify a file without including its extension, IntraBuilder assumes JFM.

<prompt expL>

If true, displays a dialog box which lets you choose between the Form Expert and the Form Designer. The default is false; you are taken directly to the Form Expert.

<home page expL>

If true, opens the Form Expert in home page mode. The default is false.

Description

Use *forms.expert*() to open the Form Expert or Form Designer and create a form interactively. The Form Expert asks a short series of questions and then builds a form which you can run immediately or bring into the Form Designer for further modification. In either case, an IntraBuilder script that defines the contents and format of a form is generated into an editable text file with a JFM extension.

When opened in home page mode, the Form Expert asks a different series of questions, ones that are used to build a home page with company information and links to other IntraBuilder forms and reports. There is no difference in the Form Designer for home pages.

Because a JFM file is a script, you can edit it with *scripts.design*().

Note The Forms Designer is a two-way tool. You can open a form in the Form Designer even if you've edited the code in the JFM file.

forms.expert() example

The following statement starts the Form Expert in Home Page mode to design the form HOME.JFM:

```
_sys.forms.expert( "HOME", false, true )
```

forms.run()

Related topics Example

Executes a form.

Syntax

_sys.forms.run(<filename expC> [,<exp1> ...])

<filename expC>

The form you want to run. You may specify “?” or a file name skeleton as the file name that will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JFM.

<exp1> ...

Optional parameters that are passed to the form script.

Description

Because a form is stored as a script file, the only difference between *forms.run*() and *scripts.run*() is that *forms.run*() assumes a JFM extension, while *scripts.run*() assumes a JS extension.

Running a form script opens the form. When a form is open on the IntraBuilder Designer, it is displayed as a separate modeless window. You can interact with the form or ignore it, and it will stay open until closed.

When a form is run by an IntraBuilder Agent, the form is displayed in the client browser that requested the form. *run*() is the only method of the ***_sys.forms*** object that you can use on an IntraBuilder Agent.

forms.run() example

The following is an *onServerClick* event handler for a button on a form. When the button is clicked, the form VIEWER.JFM is run:

```
function viewerButton_onServerClick()
{
  _sys.forms.run( "VIEWER" );
}
```

images.design()

Related topics Example

Opens a bitmap editor to create or modify an image file.

Syntax

_*sys*.*images*.*design*([<filename expC>])

<filename expC>

The image you want to create or modify. If you leave out this parameter or pass an empty string, a new untitled image is created. You may specify “?” or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes .BMP.

Description

images.*design*() opens the bitmap image editor that is registered with the operating system.

images.design() example

This example opens the Image Designer for the file IBSPLASH.BMP:

```
_sys.images.design( "IBSPLASH" )
```

images.run()

Related topics Example

Displays an image stored in a file on the IntraBuilder Designer.

Syntax

_sys.images.run(<filename expC> [,<timeout expN> [,<print expL>]])

<filename expC>

The image you want to display. You may specify “?” or a file name skeleton as the file name that will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes .BMP.

<timeout expN>

Specifies the number of seconds the image is displayed onscreen.

<print expL>

If true, sends the image to the printer as well as to the screen.

Description

Use *images.run*() to display a graphic image that was saved as a bitmap. IntraBuilder supports a number of bitmap formats, including BMP, PCX, TIF, JPG, and GIF. The image is displayed in a window.

To display an image in a client browser, use an Image component on a form or report. *images.run*() is intended to display images in the IntraBuilder Designer only.

images.run() example

images.*run*() is the method that is executed when you double-click an image in the Images tab of the IntraBuilder explorer. It is usually easier to click the image in the Explorer than it is to type out the entire *images*.*run*() statement. For example, if you double-click on the IBSPLASH.BMP file in IntraBuilder's Samples subdirectory, the following statement is streamed out to the Script Pad and executed:

```
_sys.images.run("C:\\Program Files\\Borland\\IntraBuilder\\samples\\
IBSPLASH.BMP")
```

When the statement is streamed out by the IntraBuilder Explorer, it includes the full path. If you were to type the statement yourself, you would have to type the file name only (assuming you're in the correct directory):

```
_sys.images.run( "IBSPLASH.BMP" )
```

inspect()

Example

Opens the Inspector, a window that lists object properties and lets you change their settings.

Syntax

_sys.*inspect*(<oRef>)

or

inspect(<oRef>)

<oRef>

A reference to the object you want to inspect.

Description

Use *inspect*() to examine and change object properties directly. For example, during application development you can use *inspect*() to evaluate objects and experiment with different property settings.

You can call *inspect*() directly or as a method of the ***_sys*** object.

The Inspector is modeless, and doesn't affect script execution.

Note You can access the Inspector from the Form and Report Designers by right-clicking the design surface and selecting Inspector from the shortcut menu, or by pressing F11.

You can get help on any property in the Inspector by selecting the property and pressing F1.

inspect() example

To inspect the *_sys* object and see if there are any undocumented features, type:

```
inspect( _sys )
```

os.changeDir()

Related topics Example

Changes the current default drive or directory.

Syntax

_*sys*.*os*.*changeDir*([<path expC>])

<path expC>

The new default path. For every backslash used as part of the path inside of a literal string, remember to use a double backslash (\\), since the backslash acts as the literal string escape character. You may also use a single forward slash (/) instead.

Description

Use *os*.*changeDir*() to change the current working directory to any valid drive or directory. The current directory appears in a Select control at the top of the IntraBuilder Explorer. If you change the directory in the IntraBuilder Explorer, the equivalent *os*.*changeDir*() statement is streamed out and executed in the Script Pad.

The <path expC> follows all the standard operating system rules and shortcuts regarding path names; for example,

- Path names are not case-sensitive.
- The Universal Naming Convention (UNC), starting a resource name with double backslashes (four backslashes in a literal string), is supported.
- A full path is composed of two parts: the drive, or volume, and the directory. These two may be specified together.
- If you specify a drive only, you will go to the currently selected directory on that drive.
- If you start the directory with a backslash (double backslash in a literal string), the path starts from the root directory of the drive.
- Double dots as a directory name indicate the parent directory.
- A single dot indicates the current directory; therefore using it in a path name is superfluous.
- Otherwise, the path is evaluated relative to the current drive and directory.

os.*changeDir*() also returns the resulting drive and directory. If you specify a new path, it will return the full UNC or drive and path name for the new location. For example, if you specify just a drive or just double dots, *os*.*changeDir*() will return the full drive and directory. You can call *os*.*changeDir*() with no parameters to get the current drive and directory.

Every connection serviced by an IntraBuilder Agent has its own current drive and directory, which is represented by the form or report's *virtualRoot* property. As each client request is serviced, the IntraBuilder Agent switches to that connection's *virtualRoot* directory. Once the request is finished, the IntraBuilder Agent switches back to the IntraBuilder home directory and waits for the next request. Therefore, if you switch directories during a server-side event handler with *os*.*changeDir*(), that directory will be the current directory only for the duration of that event handler. For example, if you switch directories in a form's *onServerLoad* event, that will not be the current directory when a button's *onServerClick* event fires. The current directory at the beginning of each event handler is the *virtualRoot* directory. For more information see *virtualRoot*.

os.changeDir() example

The following example switches to a subdirectory, processes some data in that directory, then switches back to the parent directory:

```
_sys.os.changeDir( "BATCH" );  // Switch to subdirectory for data processing
// Some data processing stuff
_sys.os.changeDir( ".." );     // Switch back to main directory
```

os.delete()

Example

Deletes the specified file from disk.

Syntax

_*sys*.*os*.*delete*([<filename expC>])

<filename expC>

The path/name of the file you want to delete. For every backslash used as part of the path inside of a literal string, remember to use a double backslash (\\), since the backslash acts as the literal string escape character. You may also use a single forward slash (/) instead.

Description

Use *os*.*delete*() to delete a single file from a disk.

You cannot use wildcard characters to delete multiple files with similar names. You must specify the complete file name. Like path names, file names are not case-sensitive.

If specified, the path follows all the standard operating system rules and shortcuts regarding path names, as described for the *os*.*changeDir*() method. Otherwise, the current directory is searched for the specified file.

os.delete() example

The following statement deletes the file TMP.$$$:

```
_sys.os.delete( "tmp.$$$" );
```

os.dir()

Example

Displays a disk directory listing.

Syntax

_*sys*.*os*.*dir*([<filespec expC>])

<filespec expC>

The path and/or filename(s) you want to list. If not specified, all files in the current directory are listed. For every backslash used as part of the path inside of a literal string, remember to use a double backslash (\\), since the backslash acts as the literal string escape character. You may also use a single forward slash (/) instead.

Description

Use the *os*.*dir*() method to display the contents of a directory on disk. The short (8.3) file name, file size, last modified date, and long file name for each file is displayed in the Results pane of the Script Pad. For example,

```
LOGIN.JFM          14626 08/09/1996 LOGIN.JFM
VIEWER.JFM         16303 08/09/1996 Viewer.jfm
BACKUP~1.JFM       14319 08/09/1996 Backup of LOGIN.JFM
BACKUP~2.JFM       15532 08/09/1996 Backup of VIEWER.JFM
BACKUP~3.JFM         966 08/08/1996 Backup of ANATOMY.JFM
ANATOMY.JFM          969 08/08/1996 Anatomy.jfm
62,715 bytes used in 6 files.
  131,940,352 bytes available on disk.
  527,138,816 bytes of total disk space.
```

The total of the file sizes—not the amount of disk space used—is displayed at the end of the file list, along with the number of bytes available on the disk and the total disk space.

If specified, the path follows all the standard operating system rules and shortcuts regarding path names, as described for the *os*.*changeDir*() method.

You can use the standard wildcard characters * and ? to identify multiple files with similar names. Like path names, file names are not case-sensitive.

os.dir() example

The following statement displays all the JFM files in the current directory:

```
_sys.os.dir( "*.jfm" );
```

os.diskSpace()

Example

Returns the number of bytes available on the current or specified drive.

Syntax

_*sys*.*os*.*diskSpace*([<drive expC>])

<drive expC>

The drive letter and colon (:), or the UNC volume name. If not specified, the current drive is checked.

Description

Use *os*.*diskSpace*() to determine how much space is left on a disk.

os.diskSpace() example

Suppose you want to create an administration form that you can run from anywhere with a browser. One thing you want the form to report is the amount of free disk space left on the machine running the IntraBuilder Server. You create an HTML component with the following *onServerLoad* event:

```
function diskSpaceLabel_onServerLoad()
{
  this.text = "Free disk space: " + _sys.os.diskSpace();
}
```

os.exec()

Example

Executes another Windows application, DOS application, or single DOS command.

Syntax

_sys.os.exec(<command expC> [, <DOS expL>])

<command expC>

A command or application name.

<DOS expL>

If true, runs the command as a DOS command.

Description

Use *os.exec*() to execute another application or a single DOS command. To execute multiple DOS commands interactively, you can start a DOS command window with the statement:

```
_sys.os.exec( "command", true )
```

Applications started by *os.exec*() run independently of IntraBuilder, as if they were started from the Windows Explorer.

os.exec() example

The following statement opens the Windows calculator:

```
_sys.os.exec( "calc" )
```

os.makeDir()

Related topics Example

Creates a new directory on disk.

Syntax

_sys.os.makeDir(<directory expC>)

<directory expC>

The directory you want to create.

Description

Use *os.makeDir*() to create a new directory.

The new directory name must follow the standard naming conventions for the operating system.

If you try to make a directory that already exists or is on a path that does not exist you will get an error message.

After you create the new directory, you can use *os.changeDir*() to make the new directory the current directory.

os.makeDir() example

The following example creates a directory named TMD off IntraBuilder's Apps subdirectory. It uses the *env*.*home*() method to locate the IntraBuilder home directory. That method returns the home directory with a trailing backslash, so you don't want a backslash before Apps.

```
_sys.os.makeDir( _sys.env.home() + "apps\\tmd" );
```

This example uses a working directory to process some data. If necessary, it creates the directory first. It uses a *try* block to handle any possible errors. The most likely error is that the directory already exists. By calling *os*.*makeDir*() inside a *try* block, you don't have to check if the directory already exists:

```
#define WORK_DIR "BATCH"
try {
  _sys.os.makeDir( WORK_DIR );
}
catch ( Exception e ) {
  // Do nothing on error
}
_sys.os.changeDir( WORK_DIR );
```

A manifest constant is created with the #define preprocessor directive to represent the name of the directory. If the name changes, you only have to change the #define directive, as opposed to every statement where the directory name is used.

The *os*.*changeDir*() call is not in the *try* block; if it fails, then that's a legitimate error.

protection.protect()

Example

Opens IntraBuilder's DBF table security administrator.

Syntax

_sys.*protection*.*protect*()

Description

Calling the *_sys* object's *protection*.*protect*() method opens IntraBuilder's DBF table security administrator, for historical reasons referred to as simply PROTECT.

The first time you run PROTECT, it prompts you to enter and confirm an administrator password.

Warning Remembering the administrator password is essential. You can access the security system only if you can supply the password. Once established, the security system can be changed only if you enter the administrator password when you call *protection*.*protect*(). Keep a hard copy of the database administrator password in a secured area. There is no way to retrieve a password from the system.

Once you enter the administrator password, you may setup and modify DBF table security.

protection.protect() example

To start administering DBF table security, execute the following statement in the Script Pad:

```
_sys.protection.protect()
```

queries.design()

Related topics Example

Opens the Visual Query Builder or SQL Statement Query Editor to create or modify a query.

Syntax

_sys.queries.design([<filename expC> [, <extension expC>]])

<filename expC>

The query you want to create or modify. If you leave out this parameter or pass an empty string, a new untitled query is created. You may specify "?" or a file name skeleton as the file name. This will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes .QRY.

<extension expC>

An optional string containing a file-name extension. If the string is ".SQL" (not case-sensitive), the SQL Statement Query Editor is used. In all other cases, the Visual Query Builder is used. This parameter has no effect on the default file-name extension assumed by <filename expC>.

Description

IntraBuilder supports two types of query files: QRY files and SQL files. QRY files are generated by the Visual Query Builder. SQL files are simple text files that contain an SQL statement.

The <extension expC> parameter determines which type of file is created or modified by *queries.design*(). If the parameter is the string ".SQL" then SQL files are used. The parameter is not case-sensitive; for example, you could specify the string ".sql". The parameter does not have to be a literal string; it could be a string variable or property. In all other cases—the string is not ".SQL" or the <extension expC> parameter is not specified—QRY files are used.

QRY files are created and modified with the Visual Query Builder, which generates a text file that contains the SQL statement. To create a new QRY, you must leave out the <filename expC> parameter or specify an empty string; the file-name is chosen when the QRY is saved. If you specify a <filename expC>, the Visual Query Builder will attempt to modify that file.

SQL files are created and modified with the SQL Statement Query Editor, a variation of IntraBuilder's Script Editor. You may specify a file-name when you create the SQL file. With the SQL Statement Query Editor, you type in the actual SQL statement, as opposed to designing the query visually.

queries.design() example

The following statement opens the Visual Query Builder to create a new QRY:

```
_sys.queries.design()
```

This statement opens the SQL Statement Query Editor to create the file VACATION.SQL:

```
_sys.queries.design( "vacation.sql", ".sql" )
```

The following statement modifies the file BONUS.QRY:

```
_sys.queries.design( "bonus" )
```

queries.run()

Related topics Example

Displays the results of a query in the IntraBuilder Designer.

Syntax

_*sys*.*queries*.*run*(<filename expC>)

<filename expC>

The query you want to run. You must include both the file name and extension. You may specify “?” or a file name skeleton as the file name that will display a dialog box, from which you can select a file.

Description

Use *queries*.*run*() to display the results of a query, without having to create a form first. A simple data entry form is automatically generated. If the query results in an editable rowset, the data may be edited.

queries.*run*() is intended for use in the IntraBuilder Designer only. The simple data entry form will not be sent to a client browser by the IntraBuilder Agent. To display the results of a query on a client browser, use the query in a form.

queries.run() example

The following example runs the query BIOLIFE.QRY:

```
_sys.queries.run( "biolife.qry" )
```

reports.design()

Related topics Example

Opens the Report Designer to create or modify a report.

Syntax

_sys.reports.design([<filename expC>])

<filename expC>

The report you want to create or modify. If you leave out this parameter or pass an empty string, a new untitled report is created. You may specify “?” or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JRP.

Description

Use *reports.design*() to open the Report Designer and create or modify a report interactively. The Report Designer automatically generates an IntraBuilder script that defines the contents and format of a report, and stores this code in an editable text file with a JRP extension.

When a file name is specified, the presence of that file determines whether a create or modify operation occurs. If the JRP file exists, it is modified in the Report Designer. If the file doesn't exist, it is created.

Because JRP files are scripts, you can edit them with *scripts.design*().

Note The Report Designer is a two-way tool. You can open a report in the Report Designer even if you've edited the code in the JRP file.

reports.design() example

To open a new untitled report design surface, execute the following statement in the Script Pad:

```
_sys.reports.design()
```

reports.expert()

Related topics Example

Opens the Report Expert to automatically create a report.

Syntax

_sys.reports.expert([<filename expC> [,<prompt expL>]])

<filename expC>

The name of the report you want to create. If you leave out this parameter or pass an empty string, a new untitled report is created. If you specify a file without including its extension, IntraBuilder assumes JRP.

<prompt expL>

If true, displays a dialog which allows you to choose between the Report Expert and the Report Designer.

Description

Use *reports.expert*() to open the Report Expert or Report Designer and create a report interactively. The Report Expert asks a short series of questions and then builds a report which you can run immediately or bring into the Report Designer for further modification. In either case, an IntraBuilder script that defines the contents and format of a report is generated into an editable text file with a JRP extension.

Because a JRP file is a script, you can edit it with *scripts.design*().

Note The Report Designer is a two-way tool. You can open a report in the Report Designer even if you've edited the code in the JRP file.

reports.expert() example

The following statement starts the Report Expert:

```
_sys.reports.expert()
```

reports.run()

Related topics Example

Executes a report.

Syntax

_sys.reports.run(<filename expC> [,<exp1> ...])

<filename expC>

The report you want to run. You may specify “?” or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JRP.

<exp1> ...

Optional parameters that are passed to the report script.

Description

Because a report is stored as a script file, the only difference between reports.run() and scripts.run() is that reports.run() assumes a JRP extension, while scripts.run() assumes a JS extension.

Running a report script renders the report. When a report is rendered in the IntraBuilder Designer, it is displayed in a report preview window. You can view the report and save it as static HTML.

When a report is run by an IntraBuilder Agent, that report is displayed in the client browser that requested the report. *run*() is the only method of the ***_sys.reports*** object that you can use on an IntraBuilder Agent.

reports.run() example

The following is an *onServerClick* event handler for a button on a form. When the button is clicked, the NEWMSG.JRP is run:

```
function listNewButton_onServerClick()
{
  _sys.reports.run( "NEWMSG" );
}
```

scriptOut.clear()

Related topics Example

Erases the contents of the Results pane of the Script Pad.

Syntax

_sys.scriptOut.clear([<expC>])

<expC>

Fills the Results pane of the Script Pad with the first character of <expC>. The fill character is reused on subsequent calls to *scriptOut.clear*() until another <expC> is specified. The default is to fill with spaces.

Description

Use *scriptOut.clear*() to erase the contents of the Results pane of the Script Pad. Use the option <expC> to specify a character with which to fill the pane.

The *scriptOut.column* property is set to 0 (zero) when *scriptOut.clear*() is called.

scriptOut.clear() example

The following statement clears the Results pane of the Script Pad:

```
_sys.scriptOut.clear()
```

scriptOut.column

Related topics Example

The current column position in the Results pane of the Script Pad.

Description

scriptOut.*column* reflects the current column position in the Results pane of the Script Pad. The first column is column 0 (zero). Output from the *scriptOut*.*write*() method begins at the column specified by the *scriptOut*.*column* property. *scriptOut*.*writeln*() always starts on a new line, at column 0 (zero).

The *scriptOut*.*write*() and *scriptOut*.*writeln*() methods update the *scriptOut*.*column* property to reflect the column after the last character output. You may also assign a new column number to the *scriptOut*.*column* property. The *scriptOut*.*clear*() method sets it to 0 (zero).

scriptOut.column example

The following excerpt from a script displays the contents of a table in an outline style, using one of its fields to determine the amount of indent for each row. The query has already been opened, and a reference to its rowset has been stored in the variable r:

```
r.first();
while ( !r.endOfSet ) {
  _sys.scriptOut.writeln();                              // Start each row on
new line
  _sys.scriptOut.column = 3 * r.fields[ "Level" ].value; // Indent
accordingly
  _sys.scriptOut.write( r.fields[ "Name" ].value );      // Output field
  r.next();
}
```

scriptOut.write()

Related topics Example

Displays the results of 0 or more expressions on the current line of the Results pane of the Script Pad.

Syntax

_*sys*.*scriptOut*.*write*([<exp1> [, <exp2>...]])

<exp1> [, <exp2>...]

The expressions you want to evaluate and display.

Description

Use *scriptOut*.*write*() to display the value of valid expressions of any type. If called with no parameters, *scriptOut*.*write*() has no effect.

The *scriptOut*.*write*() method is identical to the *scriptOut*.*writeln*() method, except it displays on the current line starting at the current column position, as reflected by the *scriptOut*.*column* property, rather than on a new line.

The *scriptOut*.*column* property is updated to reflect the column after the last character displayed.

scriptOut.write() example

The following excerpt from a script displays a running counter as it processes rows in a table. The query has already been opened, and a reference to its rowset has been stored in the variable r:

```
r.first();
_sys.scriptOut.writeln( "Rows processed:" );  // Display label
var nCol = _sys.scriptOut.column;             // Remember column after label
var nRow = 0;                                 // Initialize table row counter
while( !r.endOfSet ) {
  ++nRow;                                     // Increment counter for each
row
  // Some processing
  if ( nRow % 100 == 0 ) {                    // For every hundredth row
    _sys.scriptOut.write( nRow  );            //   Display rows so far
    _sys.scriptOut.column = nCol;             //   and go back to column on
line
  }
  r.next();
}
_sys.scriptOut.write( nRow  );                //   Display final row count
```

scriptOut.writeln()

Related topics Example

Displays the results of 0 or more expressions on a new line in the Results pane of the Script Pad.

Syntax

_*sys*.*scriptOut*.*writeln*([<exp1> [, <exp2>...]])

<exp1> [, <exp2>...]

The expressions you want to evaluate and display.

Description

Use *scriptOut*.*writeln*() to display the value of valid expressions of any type. *scriptOut*.*writeln*() starts a new line in the Results pane of the Script Pad before displaying values, if any.

The *scriptOut*.*writeln*() method is identical to the *scriptOut*.*write*() method, except it starts displaying on a new line, rather than the current line.

The *scriptOut*.*column* property is updated to reflect the column after the last character displayed.

In the Script Pad, you may use the ? symbol as shorthand for the *scriptOut*.*writeln*() method.

scriptOut.writeln() example

The following statement from a script displays the current date and time:

```
_sys.scriptOut.writeln( new Date() )
```

The same statement can be executed in the Script Pad as:

```
? new Date()
```

scripts.compile()

Related topics Example

Compiles a script file into byte code.

Syntax

_sys.scripts.compile(<filespec1 expC> [,<filespec2 expC> ...])

<filespec1 expC> [, <filespec2 expC> ...]

The script(s) you want to compile. If you specify a file without including its extension, IntraBuilder assumes JS. You may use standard wildcard characters * and ? in the file-name specification.

Description

In order to run a script in IntraBuilder, it must be compiled into byte code first. When you attempt to run a source (text) file, like a JS or JFM file, IntraBuilder compares the time stamp of that file with its byte code equivalent. If the source file is more recent, the file is automatically compiled before it is run.

You may also compile the file manually with the *scripts.compile*() method, which compiles the file without executing it.

When you compile a file, the resulting byte code is stored in a file with the same name and extension, but with the last letter of the extension changed to the letter O: JS becomes JO, JFM becomes JFO, and so on.

A Compilation Status dialog box displays compilation statistics for the current file and totals for all the files compiled in the *scripts.compile*() call.

scripts.compile() example

The following statement compiles the script file QWK.JS:

```
_sys.scripts.compile( "QWK" )
```

This statement compiles the form script file VIEWER.JFM:

```
_sys.scripts.compile( "VIEWER.JFM" )
```

The following statement compiles all the script, form, and report files in the current directory:

```
_sys.scripts.compile( "*", "*.jfm", "*.jrp" )
```

scripts.delete()

Related topics Example

Deletes a script file and any associated byte code file.

Syntax

_*sys*.*scripts*.*delete*([<filename expC>])

<filename expC>

The script file you want to delete. You must include the file-name extension.

Description

Use *scripts*.*delete*() to delete a script file and its associated byte code file, if any.

When you compile a file, the resulting byte code is stored in a file with the same name and extension, but with the last letter of the extension changed to the letter O: JS becomes JO, JFM becomes JFO, etc.

scripts.delete() example

The following statement will delete the script file TEST.JFM and its associated byte code file TEST.JFO, if it exists:

```
_sys.scripts.delete( "TEST.JFM" )
```

scripts.design()

Related topics Example

Opens the Script Editor to create or modify a script.

Syntax

_*sys*.*scripts*.*design*([<filename expC>])

<filename expC>

The script you want to create or modify. If you leave out this parameter or pass an empty string, a new untitled script is created. You may specify "?" or a file name skeleton as the file name that will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JS.

Default

By default, *scripts*.*design*() launches IntraBuilder's internal Script Editor. You can designate an alternate editor in the Desktop Properties dialog box.

Description

Use *scripts*.*design*() to open the Script Editor and create or modify a script.

If the script file is loaded into memory, it is implicitly unloaded when you edit it.

scripts.design() example

The following statement opens a new untitled Script Editor:

```
_sys.scripts.design()
```

scripts.load()

Related topics Example

Loads an IntraBuilder script into memory, making its functions, classes, and methods available for execution.

Syntax

_sys.scripts.load(<filename expC>)

<filename expC>

The script you want to load. You may specify "?" or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JS.

Description

To execute a function or method, that function must be loaded in memory. To be more precise, a simple pointer to that function must be in memory. The contents of the function itself are not necessarily in memory at any given time; if not, the contents get loaded into memory automatically when the function is executed. But if that function's pointer is in memory, it is considered to be loaded.

Whenever you execute a script file with the *scripts.run*() method, it is loaded implicitly; pointers to all of the functions, classes, and methods in that file are loaded into memory. Therefore, code in a script file may always call any other functions or methods in the same file.

To access functions, classes, and methods in other script files, load the script file with the *scripts.load*() method first. Its function pointers stay in memory until the script file is unloaded with the *scripts.unload*() method, or until the script file is edited with the *scripts.design*() method.

IntraBuilder uses a reference count system to manage script files in memory. Each loaded script file has a counter for the number of times it has been loaded, either explicitly with *scripts.load*() or implicitly. As long as the count is more than zero, the file stays loaded. Calling *scripts.unload*() reduces the count by one. Therefore, if you call *scripts.load*() twice, you need to call *scripts.unload*() twice to remove the script from memory. Editing the script with *scripts.design*(), even if you don't make any changes, always removes the script from memory, no matter what the reference count is.

A script's load count has no impact on memory; it is simply a counter. Loading a script 10 times uses the same amount of memory as loading it once.

Whenever a function is called, the function pointers for the file that contains the currently executing script, if any, are searched first. Secondarily, if two functions with the same name in two separate script files are loaded into memory, the function that was loaded first is used when that function name is called.

scripts.load() example

The following statement loads the functions, classes, and methods in the script file LIBRARY.JS into memory:

```
_sys.scripts.load( "LIBRARY" )
```

scripts.run()

Related topics Example

Executes a script.

Syntax

_sys.scripts.run(<filename expC>)

<filename expC>

The script you want to execute. You may specify “?” or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JS.

Description

Use *scripts.run*() to run a script file. All script statements outside of functions and classes are executed from the top of the file down.

Whenever you run a script file, it is loaded implicitly; pointers to all of the functions, classes, and methods in that file are loaded into memory. Therefore, code in a script file may always call any other functions or methods in the same file.

In order to run a script in IntraBuilder, it must be compiled into byte code first. When you specify a source (text) file, like a JS or JFM file, IntraBuilder compares the time stamp of that file with its byte code equivalent. If the source file is more recent, the file is automatically compiled before it is run.

scripts.run() example

This example runs a script named TEST.JS:

```
_sys.scripts.run( "TEST" )
```

scripts.unload()

Related topics Example

Unloads an IntraBuilder script from memory, preventing further access and execution of its functions, classes, and methods.

Syntax

_*sys*.*scripts*.*unload*(<filename expC>)

<filename expC>

The script you want to unload. You may specify “?” or a file name skeleton as the file name which will display a dialog box, from which you can select a file. If you specify a file without including its extension, IntraBuilder assumes JS.

Description

scripts.*unload*() reduces the load count of the specified script file by one. If that reduces its load count to zero, then that script file is removed from memory.

See *scripts*.*load*() for a description of the reference count system used to manage script files.

scripts.unload() example

The following statements load a script file into memory, call one of its functions, and unload the file:

```
_sys.scripts.load( "QWK.JS" );
genQWK();
_sys.scripts.unload( "QWK.JS" );
```

tables.design()

Related topics Example

Opens the Table Designer to create or modify a table.

Syntax

_*sys*.tables.design([<name expC> [, <type expC>]])

<name expC>

The table you want to create or modify. If you leave out this parameter or pass an empty string, a new untitled table is created. You may specify "?" or a name skeleton as the name, which will display a dialog box from which you can select a table.

When using the Standard table driver, if you specify a file name without including its extension, IntraBuilder assumes DBF or DB, depending on the table type. If you specify a DBF or DB extension, the type becomes dBASE or Paradox, respectively.

<type expC>

A string designating the table type, "dBASE" or "Paradox" (not case-sensitive). The default is configured as part of the Borland Database Engine. This optional parameter has an effect only when using the Standard table driver.

Description

Use *tables*.*design*() to open the Table Designer and create or modify a table structure.

tables.design() example

The following statement opens the Table Designer to create a new Paradox table:

```
_sys.tables.design( "", "paradox" )
```

tables.expert()

Related topics Example

Opens the Table Expert to automatically create a table.

Syntax

_*sys*.tables.expert([<name expC> [,<prompt expL>]])

<name expC>

The name of the table you want to create. If you leave out this parameter or pass an empty string, a new untitled table is created.

<prompt expL>

If true, displays a dialog which allows you to choose between the Table Expert and the Table Designer.

Description

Use *tables*.*expert*() to open the Table Expert or Table Designer and create a table structure. The Table Expert provides a number of sample tables from which you can borrow fields, then builds a table that you can run immediately or bring into the Table Designer for further modification.

tables.expert() example

The following statement opens the Table Expert to create the table FONELIST.DBF:

```
_sys.table.expert( "fonelist" );
```

tables.run()

Related topics Example

Opens a table for editing in the IntraBuilder Designer.

Syntax

_sys.tables.run(<name expC>)

<name expC>

The table you want to edit. You may specify “?” or a name skeleton as the name, which will display a dialog box from which you can select a table.

Description

Use *tables.run*() to display and edit the contents of a table, without having to create a form first. A simple data entry form is automatically generated.

tables.run() is intended for use in the IntraBuilder Designer only. The simple data entry form will not be sent to a client browser by the IntraBuilder Agent.

tables.run() example

The following statement displays the contents of the table SECTIONS.DB:

```
_sys.tables.run( "sections.db" )
```

Preprocessor

When IntraBuilder compiles a JavaScript script file, that file is run through the preprocessor before it is actually compiled. The preprocessor is a separate built-in utility that processes the text of the script file to prepare it for compilation. Its duties include

- Stripping out comments from the script file
- Substituting macro-identifiers and macro-functions with the corresponding replacement text
- Including the text of other files in the script file
- Conditionally excluding parts of the script file so they are not compiled

The preprocessor generates an intermediate file; this is the file that is compiled by IntraBuilder's JavaScript compiler.

While those are the mechanics of the preprocessor, the usage of the preprocessor allows you to

- Replace constants and "magic numbers" in your code with easy-to-read and easy-to-change identifiers
- Create macro-functions to replace complex expressions with parameters
- Use collections of constant identifiers and macro-functions in multiple script files
- Maintain separate versions of your scripts, for example debug and production versions, in the same script files through conditional compilation

IntraBuilder's preprocessor is similar to the preprocessor used in the C language. It uses a handful of preprocessor directives to control its activities. All preprocessor directives start with the # character and each one must be on its own, single line in the script file. They do not use a line termination character.

#define

Related topics Example

Defines an identifier (name) for use in controlling script compilation, defining constants, or creating macro-functions.

Syntax

#define <identifier> [<replacement text>]

#define <identifier>(<parameter list>) <replacement text with parameters>

<identifier>

A name. It identifies the text to replace if <replacement text> is supplied. The name must start with an alphabetic character and can contain any combination of alphabetic or numeric characters, uppercase or lowercase letters. The identifier is case-sensitive.

(<parameter list>)

Parameter names that correspond to arguments passed to a macro-function that you create with #define <identifier>(<parameter list>) <replacement text>. If you specify multiple parameters, separate each with a comma. There cannot be any spaces between the <identifier> and the opening parenthesis of (<parameter list>), or after any of the parameter names in the <parameter list>.

<replacement text>

The text you want to use to replace all occurrences of <identifier>. If you specify <replacement text>, the preprocessor scans each source code line for identifiers and replaces each one it encounters with the specified replacement text. <replacement text> can be any text that is part of a JavaScript script, such as a string, numeric expression, or series of statements.

Description

The #define directive defines an identifier and optionally lets you replace text in a script before compilation. Each #define definition must begin on a new line and is limited to 4096 characters.

Identifiers are available only to the script in which they are defined. To define identifiers for use in multiple scripts, place them in a separate file and use #include to include that file as needed.

You must define an identifier in a file with the #define directive before you can use it. Once it has been defined, you cannot #define it again; you must undefine it first with the #undef preprocessor directive.

Use the #define directive for the following purposes:

- To declare an identifier and assign replacement text to represent a constant value or a complex expression.
- To create a macro-function.
- To declare an identifier with no replacement text, so you can use it with the #ifdef or #ifndef directive.
- To declare an identifier with replacement text, so you can use it with the #if directive.

The effect of #define is similar to a word processor's search-and-replace feature. When the preprocessor encounters a #define identifier in the text of a script, it replaces that identifier with the <replacement text>. If there is no <replacement text> for that identifier, the identifier is simply removed. For example:

```
#define test 4                        // Create identifier with value
_sys.scriptOut.writeln( test - 3 );   // ( 4 - 3 ) = 1
#undef test                           // #undef to change definition
#define test                          // Create identifier with no value
_sys.scriptOut.writeln( test - 3 );   // ( - 3 ) = -3
```

Because the preprocessor runs before a script is compiled and performs simple text substitution, the use of #define statements can in effect override variables, built-in functions, and any other element having the same name as <identifier>. This is shown in the following examples.

```
// Overiding a variable
somevar = 25;                     // Creates variable
```

```
#define somevar 10;               // Until further notice, "somevar" will be
replaced
_sys.scriptOut.writeln( somevar ) // Compiles argument as "10". Displays 10
#undef somevar                    // "somevar" no longer replaced
_sys.scriptOut.writeln( somevar ) // Displays 25
// Overriding a function
#define parseInt(x) (42 + x)           // Function adds 42
_sys.scriptOut.writeln( parseInt(3) ); // Compiles argument as "(42 + 3)".
Displays 45
```

To use #define directives in JFM and JRP files generated by the Form and Report Designer, place the directives in the Header section of the file so that the definitions will not be erased by the Designer.

Declaring identifiers to represent constants

Assign an identifier to represent a constant value or expression when you want the preprocessor to search for and replace all instances of the identifier with the specified value or expression before compilation. When used in this manner, the identifier is known as a manifest constant. It's common practice to make the name of the manifest constant all uppercase, with underscores between words so that it stands out in code. For example:

```
#define SECS_PER_HOUR 3600                     // Number of seconds per hour
#define MSECS_PER_DAY (1000*24*SECS_PER_HOUR)  // Number of milliseconds per
day
```

Note that when using a manifest constant to represent a numeric expression, you should place the entire expression inside parentheses. This prevents possible errors due to the precedence of operators used to evaluate expressions. For example, consider the following calculation:

```
nDays = nTimeElapsed / MSECS_PER_DAY;
```

Without parentheses, this statement would compile as:

```
nDays = nTimeElapsed / 1000*24*3600;
```

That's incorrect—it divides by 1000 then multiplies by 24 and 3600. (The multiplication and division operators are at the same level of precedence, so the expression is evaluated from left to right.). By placing parentheses around the manifest constant definition as shown, the statement would compile as:

```
nDays = nTimeElapsed / (1000*24*3600);
```

Because of the parentheses, the expression is evaluated correctly: the value of the constant is evaluated first, then used as the divisor.

Manifest constants streamline your code and improve its readability because you can use a single identifier to represent a frequently used constant or a complex expression. In addition, if you need to change the value of a constant in your script, you need to change only the constant definition and not every occurrence of the constant.

To replace an identifier only in parts of a script, insert #undef <identifier> into your script where you want the search-and-replace process to stop.

Creating macro-functions

When the preprocessor encounters a function call that matches a defined macro-function, it replaces the function call with the replacement text, inserting the arguments of the function call into the replacement text. This is shown in the following example.

```
#define avg(num1,num2) (((num1)+(num2))/2)  // Average two numbers
Ä
n1=20;
n2=40;
_sys.scriptOut.writeln( avg( n1, n2 ) );  // Displays 30
```

The arguments in the macro-function call are substituted exactly as they are shown in the macro-function definition. In this example, the last statement compiles as:

```
_sys.scriptOut.writeln( (((n1)+(n2))/2) );
```

When using a macro-function to perform calculations, always use parentheses to enclose each

parameter and the entire expression in the macro-function definition as shown. If you leave them out, errors may result due to the precedence of operators, as shown in these (somewhat contrived) examples:

```
#define so(x) _sys.scriptOut.writeln(x)    // Less typing!
#define avg(num1,num2) (((num1)+(num2))/2)
#define badAvg(num1,num2) (num1+num2)/2
so(    avg( 2, 3 << 1 ) ); // (2+6)/2      --> displays 4
so( badavg( 2, 3 << 1 ) ); // ((2+3)<<1)/2 --> displays 5
so( 12 /    avg( 2, 4 ) ); // 12/(6/2)     --> displays 4
so( 12 / badavg( 2, 4 ) ); // 12/6/2       --> displays 1
```

Unlike functions in JavaScript, the number of arguments passed from a macro-function call must match the number of parameters defined in your #define statement.

Declaring identifiers for conditional compilation

In addition to using identifiers for constants and macro-functions in JavaScript code, they are used for conditional compilation with the #if, #ifdef, and #ifndef directives.

Defining an identifier without replacement text lets you use it with the #ifdef or #ifndef directive to test if the identifier exists. When used in this manner, the existence of the identifier acts as a logical flag to either include or exclude code for compilation.

When an identifier is defined with replacement text, you can use comparison operators to check the value of the identifier in an #if directive, and conditionally compile code based on the result. And you can still use #ifdef and #ifndef.

Nesting preprocessor identifiers

You can nest preprocessor identifiers; that is, the replacement text for one identifier may contain other identifiers, as long as those identifiers are already defined, as shown in the following example:

```
#define SECS_PER_HOUR 3600                     // Number of seconds per hour
#define MSECS_PER_DAY (1000*24*SECS_PER_HOUR)  // Number of milliseconds per
day
```

You cannot use the identifier being defined in its replacement text, however.

#define example

The first example uses a manifest constant to represent a “magic number.” Suppose your application is doing metric conversions. Instead of sprinkling the conversion factor around in your code, which would just be a cryptic number, you can #define it as a manifest constant, which eliminates the possibility that you might get the number wrong in some places and makes your code self-documenting:

```
#define LB_PER_KG 2.2046   // Number of pounds per kilogram
Ä
nPounds = this.form.kg.value * LB_PER_KG;
```

The second example uses a manifest constant to represent a simple constant in your application. Suppose you’re testing several different techniques to see which one accomplishes the same task the fastest. You need to repeat the task many times to get measurable results, so you use a manifest constant to represent the number of times you want each test to be run. By using a single manifest constant, you can easily change the number of times each test is run and calculate the average time:

```
#define NUM_REPS 10000   // Number of times to repeat each test
Ä
for ( n = 1; n <= NUM_REPS; n++ ) {
  // Test 1
}
for ( n = 1; n <= NUM_REPS; n++ ) {
  // Test 2
}
Ä
_sys.scriptOut.writeln( "Average time for test 1", time[0] / NUM_REPS );
```

The following example uses a manifest constant for a file name that is used in different parts of an application:

```
#define QWK_FILE     "IMF.QWK"
#define MESSAGE_FILE "MESSAGES.DAT"
Ä
var fMsg = new File();
fMsg.create( MESSAGE_FILE );
Ä
var z = new ZipFile( QWK_FILE );  // Create compressed file
z.store( MESSAGE_FILE );          // Store the message file
Ä
class ZipFile( cFileName ) extends File
{
  // Code to implement ZipFile class
}
```

In this example, a macro-function is used as shorthand for a verbose method call. Note that the #define does not end with a semicolon, but the use of the macro-function does:

```
#define so(x) _sys.scriptOut.writeln(x)
Ä
so( "Test" );
```

This example demonstrates conditional compilation. Two preprocessor identifiers are used: a DEBUG flag, and a BUILD number:

```
#define DEBUG     // Comment out if not debug version
#define BUILD 35  // Current build number
Ä
#if BUILD < 20
  // Older code
#else
  // Current code
  #ifdef DEBUG
```

```
    // Include DEBUG code
  #endif
#endif
```

#else

Related topics

Designates an alternate block of code to conditionally compile if the condition specified by an #if, #ifdef, or #ifndef directive is false.

#endif

Related topics

Designates the end of an #if, #ifdef, or #ifndef directive.

#if

Related topics Example

Controls conditional compilation of code based on the value of an identifier assigned with #define.

Syntax

#if <condition>
<statements 1>
[#else
<statements 2>]
#endif

<condition>

A logical expression, using an identifier you've defined, that evaluates to true or false.

<statements 1>

Any number of statements and preprocessor directives. These lines are compiled if <condition> evaluates to true.

#else <statements 2>

Specifies the lines you want to compile if <condition> evaluates to false.

Description

Use the #if directive to conditionally compile sections of source code based on the value of <identifier>. Two other directives, #ifdef and #ifndef, are also used to conditionally include or exclude code for compilation. Unlike the #if directive, however, they test only for the existence of an identifier, not for its value.

The <condition> must be a simple logical condition; that is, a single test using one basic comparison operator (==, <, >, <=, >=, !=). You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same script, for debugging, and for managing the use of #include files. Using #if for conditional compilation is different than conditionally executing code with an *if* statement. With *if*, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #if to exclude code you don't want for a particular version of your script, the code is never compiled into byte code.

When IntraBuilder's preprocessor processes a file, it internally defines the preprocessor identifier __intrabuilder__ (two underscores on both ends) with the current version number. Use this built-in identifier to manage code that's intended to run on different versions of IntraBuilder.

#if example

The following example demonstrates how you would create code that runs on different versions of IntraBuilder, using the built-in identifier __intrabuilder__:

```
#if __intrabuilder__ == 1
  // Version 1.0 code
#else
  #if __intrabuilder__ == 2
    // Version 2.0 code
  #endif
#endif
```

Because code that is excluded by #if is never compiled, you can safely use new syntax that might be introduced in a new version. When compiled with an older version of IntraBuilder, the new code is ignored. This is different than testing the version returned by the method *_sys.env.version*() at run time. New syntax would not compile under an older version.

#ifdef

Related topics Example

Controls conditional compilation of code based on the existence of an identifier created with #define.

Syntax

#ifdef <identifier>
<statements 1>
[#else
<statements 2>]
#endif

<identifier>

The identifier you want to test for. <identifier> is defined with the #define directive.

<statements 1>

Any number of statements and preprocessor directives. These lines are compiled if <identifier> has been defined.

#else <statements 2>

Specifies the lines to compile if <identifier> has not been defined.

Description

Use the #ifdef directive to conditionally compile sections of source code. If you've defined <identifier> with #define, the code you specify with <statements 1> is compiled; otherwise, the code following #else, if any, is compiled.

You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same script, for debugging purposes, and for managing the use of #include files. Using #ifdef for conditional compilation is different than not executing code with an *if* statement. With *if*, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #ifdef to exclude code you don't want for a particular version of your script, the code is never compiled into byte code.

#ifdef example

The following example uses #ifdef to check for a identifier named DEBUG to determine if extra code should be included to display trace information in the Result pane of the Script Pad:

```
#define DEBUG     // Comment out if not debug version
#ifdef DEBUG
  #define TRACE(m) _sys.scriptOut.writeln(m)
#else
  #define TRACE(m)
#endif
Ä
// Process names in list
this.form.rowset.first();
while ( !this.form.rowset.endOfSet ) {
  TRACE( this.form.rowset.fields[ "Last name" ].value );  // Display name as
we go
  // Do whatever
  this.form.rowset.next();
}
```

The macro-function TRACE() is defined so that if the DEBUG identifier is not defined, all calls to TRACE() are replaced with nothing—they are removed from the code and not compiled. This allows you to use TRACE() as much as d you want, and with a simple change in the DEBUG identifier, remove all the code from the compiled byte code, resulting in better performance.

#ifndef

Related topics Example

Controls conditional compilation of code based on the existence of an identifier assigned with #define.

Syntax

#ifndef <identifier>
<statements 1>
[#else
<statements 2>]
#endif

<identifier>

The identifier you want to test for. <identifier> is defined with the #define directive.

<statements 1>

Any number of statements and preprocessor directives. These lines are compiled if <identifier> has not been defined.

#else <statements 2>

Specifies the lines to compile if <identifier> has been defined.

Description

Use the #ifndef directive to conditionally compile sections of source code. If you haven't defined <identifier> with #define, the code you specify with <statements 1> is compiled; otherwise, the code following #else, if any, is compiled.

Use #ifndef if you want to include code only if the identifier is not defined. Otherwise, you can use #ifdef to include code only if the identifier is defined, and #ifdef with its #else option to include different sets of code depending on the existence of the identifier.

You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same script, for debugging purposes, and for managing the use of #include files. Using #ifndef for conditional compilation is different than not executing code with an *if* statement. With *if*, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #ifndef to exclude code you don't want for a particular version of your script, the code is never compiled into byte code.

#ifndef example

When creating a set of #define directives in an #include file, enclose the entire set inside an #ifndef block and #define a special identifier for that block. For example, here are some lines from the INTRA.H file that includes #define directives for many of the enumerated values used in IntraBuilder:

```
#ifndef INTRA_H
#define INTRA_H
// filterOptions property
#define FILTEROPTIONS_MATCH_LENGTH_AND_CASE      0
#define FILTEROPTIONS_MATCH_PARTIAL              1
#define FILTEROPTIONS_IGNORE_CASE                2
#define FILTEROPTIONS_MATCH_PARTIAL_IGNORE_CASE  3
#endif
```

If you #include the same file twice in the same script file (which often happens because some #include files #include other files), the #ifndef directive will make sure that #define directives are processed only once. Attempting to #define the same identifier twice causes an error.

#include

Related topics Example

Inserts the contents of the specified source file (known as an include or header file) into the current script file at the location of the #include statement.

Syntax

#include <filename> | "<filename>"

<filename> | “<filename>”

The name of the file, optionally including a full or partial path, whose contents are to be inserted into the current script file. You can specify the file name within or without quotes. An include file typically has an .h file-name extension.

If you specify <filename> without a path, the preprocessor uses the following search order:

1. It searches the current directory for the file exactly as you’ve specified it.
2. If you omitted the .h file-name extension, it adds the extension and searches the current directory.
3. If it can’t find the file in the current directory, it looks in <home directory>\ INCLUDE. (The home directory is the one returned by *_sys.env.home*().)
4. If it can’t find the file in the current directory or <home directory>\INCLUDE, it looks in the directory you specify with the DOS environment variable INCLUDE.

Description

The effect of #include is as if the contents of the specified file were typed into the current script file at the location of the #include statement. The specified file is called an include file. #include is used primarily for files which have #define directives.

Identifiers are available only to the script in which they are defined. To use a single set of identifiers in multiple scripts, save the #define statements in a file, then use the #include directive to define the identifiers in additional scripts.

An advantage of having all the #define statements in one file is the ease of maintenance. If you need to modify any of the #define statements, you need only change the include file; the script files that use the #define statements remain unchanged. After you modify the include file, recompile your script file for the changes to take effect.

To use #include directives in JFM and JRP files generated by the Form and Report Designer, place the directives in the Header section of the file so that the definitions will not be erased by the Designer.

#include example

You may want to set up a standard include file that you use in all your script files that contains manifest constants and macro-functions that you use through your application. For example:

```
// Std.h
#define so(x) _sys.scriptOut.writeln(x)  // Shorthand; used for debugging
#include "INTRA.H"                       // Contains #defines for enums
```

Place the STD.H file in IntraBuilder's INCLUDE subdirectory so that it's easily accessible. Then at the top of every script, #include that file:

```
#include "STD.H"
Ä
so( "Test" );
```

#undef

Related topics Example

Removes the current definition of the specified identifier previously defined with #define.

Syntax

#undef <identifier>

<identifier>

The identifier whose definition you want to remove.

Description

The #undef directive removes the definition of an identifier previously defined with the #define directive. If you use #define with <replacement text>, the preprocessor replaces all instances of the identifier with the replacement text from the point it encounters that #define until it encounters an #undef specifying the same identifier. Therefore, to replace an identifier only in parts
of a script, insert #undef <identifier> into your script where you want the search-and-replace process to stop.

#undef is also required if you want to change the <replacement text> for an identifier. You cannot use #define for an identifier that is already defined. You must #undef the identifier first, then specify a new #define directive.

Attempting to #undef an identifier that is not defined has no effect; no error is generated.

#undef example

In this example, the script file has numerous #ifdef DEBUG statements to conditionally compile debug code. You want to use the debug code for only one section in the file, so you #define DEBUG at the beginning of the section, and #undef DEBUG at the end:

```
Ä
#define DEBUG
// Some code
#ifdef DEBUG
  // Debug code
#endif
// More code
#undef DEBUG
Ä
// Some more code
#ifdef DEBUG
  // More debug code
#endif
```

__intrabuilder__

Related topics Example

Identifies the current IntraBuilder version number.

Description

When IntraBuilder's preprocessor processes a file, it internally defines the preprocessor identifier __intrabuilder__ (two underscores on both ends) with the current version number. Use this built-in identifier to manage code that's intended to run on different versions of IntraBuilder.

__intrabuilder__ example

The following example demonstrates how you would create code that runs on different versions of IntraBuilder, using the built-in identifier __intrabuilder__:

```
#if __intrabuilder__ == 1
  // Version 1.0 code
#else
  #if __intrabuilder__ == 2
    // Version 2.0 code
  #endif
#endif
```

Because code that is excluded by #if is never compiled, you can safely use new syntax that might be introduced in a new version. When compiled with an older version of IntraBuilder, the new code is ignored. This is different than testing the version returned by the method *_sys.env.version*() at run time. New syntax would not compile under an older version.

class NetInfo

Related topics

Information about the requesting network connection.

Syntax

[<oRef> =] new NetInfo()

<oRef>

A variable or property in which to store a reference to the newly created NetInfo object.

Properties

The following tables list the properties of the NetInfo class. (No events or methods are associated with this class.)

Property	Default	Description
className	NetInfo	Identifies the object as an instance of the NetInfo class
IPAddress	-1	IP address of client browser
referrer		URL of document that contained link to current location
remoteHost		Host name of client browser
serverName		Host name of HTTP server
sessionID	-1	IntraBuilder session number
userAgent		Client browser identification string

Events	Parameters	Description

None

Methods	Parameters	Description

None

Description

Use a NetInfo object to get information about the current network connection between the client browser and the IntraBuilder Agent.

When a NetInfo object is created in the IntraBuilder Agent, its properties reflect the connection between the Agent and the requesting client browser.

When a NetInfo object is created in the IntraBuilder Designer, its properties contain the default, empty values.

Some properties are not fully supported by all client browsers or HTTP servers.

class NetInfo example

The following *onServerLoad* event handler records all the information about the current connection in a table that has been opened with the query qLogFile.

```
function Form_onServerLoad()
{
  var ni = new NetInfo();
  var r  = this.qLogFile.rowset;
  r.beginAppend();
  r.fields[ "IP address"  ].value = ni.IPAddress;
  r.fields[ "Referrer"    ].value = ni.referrer;
  r.fields[ "Remote host" ].value = ni.remoteHost;
  r.fields[ "Server name" ].value = ni.serverName;
  r.fields[ "Session ID"  ].value = ni.sessionID;
  r.fields[ "User agent"  ].value = ni.userAgent;
  r.save();
}
```

IPAddress

Related topics

IP address of the client browser.

Property of

NetInfo

Description

IPAddress contains the 32-bit IP (Internet Protocol) address of the client browser. It is represented as a 32-bit signed number for ease and efficiency in storage and comparison. To convert the number to the dot-delimited four-byte IP address, use the following function:

```
function stringIPAddress( nIPA )
{
  return "" +
         ( nIPA & 255 ) + "." +
         (( nIPA >> 8 ) & 255 ) + "." +
         (( nIPA >> 16 ) & 255 ) + "." +
         (( nIPA >> 24 ) & 255 );
}
```

For example, *IPAddress* might contain the number 257124815. stringIPAddress(257124815) yields the string "207.105.83.15".

In some situations, the host name of the client browser is available in the *remoteHost* property. Sometimes the *remoteHost* is the dot-delimited four-byte representation of the IP address.

referrer

URL of the document that contained the link to the current location.

Property of

NetInfo

Description

If the current IntraBuilder form or report was displayed by clicking a link on another page, the *referrer* property contains the URL of that page.

Some browsers do not support this capability, so the *referrer* property might be blank.

remoteHost

Related topics

Host name of the client browser.

Property of

NetInfo

Description

If available, *remoteHost* contains the host name of the client browser. The host name is either a series of dot-separated domain names, or four dot-separated numbers that represent the IP address.

Some browsers do not support this capability, so the *remoteHost* property might be blank.

serverName

Host name of the HTTP server.

Property of

NetInfo

Description

serverName contains the host name of the HTTP (Web) server. The host name is either a series of dot-separated domain names, or four dot-separated numbers that represent the IP address.

serverName example

sessionID

Related topics

IntraBuilder session identification number.

Property of

NetInfo

Description

The IntraBuilder Broker assigns a unique ID number to every connection that it services. Use the *sessionID* property to differentiate between simultaneous connections.

For example, suppose you have a shopping cart application, and all selections from all users are stored in a single table. You store the value of the *sessionID* property in a field to identify who picked what.

sessionID is more discriminating than *IPAddress*. The IntraBuilder Agent can be configured to allow multiple connections from the same IP address. In that case, the *IPAddress* property would be the same, while the *sessionID* would be different. IP addresses can also be pooled, as they are for many large Internet service providers, which could lead to the same IP address used by two different connections in a short period of time. The *sessionID* would be different for two such connections.

userAgent

Related topics

Client browser identification string.

Property of

NetInfo

Description

Most client browsers will return an identification string that describes itself. For example, Netscape Navigator 3.0 running on Windows 95 would return something like this:

```
Mozilla/3.0 (Wind95; I)
```

and Microsoft Internet Explorer 3.0 would be something like this:

```
Mozilla/2.0 (compatible; MSIE 3.0; Windows 95)
```

Use the *userAgent* property to perform browser-specific activities.

alert()

Related topics Example

Displays an alert dialog box.

Syntax

[<oRef>.]*alert*(<expC>)

<oRef>

An object reference to a browser window.

<expC>

The text you want to display.

Property of

Window

Description

Use *alert*() to display a message in a dialog box. <expC> is displayed in the dialog box with an OK button.

Do not call *alert*() from an *onFocus* event handler. The dialog box will get focus; then when the dialog box is dismissed, the component will regain focus, and its *onFocus* event handler will fire again, resulting in an infinite loop.

The scoping rules in client-side JavaScript do not require specifying a reference to the Window object that displays the dialog box.

Although it's intended as a client-side method, *alert*() works in the IntraBuilder Designer. There are no Window objects in IntraBuilder; simply call *alert*() as a function. The dialog box is displayed in the IntraBuilder Designer.

Calling *alert*() in server-side code in the IntraBuilder Agent has no effect.

alert() example

The following statement displays a message in a dialog box.

```
alert( "Done!" );
```

confirm()

Related topics Example

Displays a confirmation dialog box and returns *true* or *false*.

Syntax

[<oRef>.]*confirm*(<expC>)

<oRef>

An object reference to a browser window.

<expC>

The text to display.

Property of

Window

Description

Use *confirm*() to display a message in a dialog box and get an OK/Cancel response. <expC> is displayed in a dialog box with OK and Cancel buttons. Clicking OK causes *confirm*() to return *true*; clicking Cancel causes *confirm*() to return *false*.

Do not call *confirm*() from an *onFocus* event handler. The dialog box will get focus; then when the dialog box is dismissed, the component will regain focus, and its *onFocus* event handler will fire again, resulting in an infinite loop.

The scoping rules in client-side JavaScript do not require specifying a reference to the Window object that displays the dialog box.

Although it's intended as a client-side method, *confirm*() works in the IntraBuilder Designer. There are no Window objects in IntraBuilder; simply call *confirm*() as a function. The dialog box is displayed in the IntraBuilder Designer.

Calling *confirm*() in server-side code in the IntraBuilder Agent has no effect; it always returns *false*.

confirm() example

The following statements use the return value from a confirmation dialog box to determine whether some code is executed.

```
if ( confirm( "This will purge all backup files" ) ) {
  // some code
}
```

File|New|Table

Create a new table by defining its structure in either the Table Expert or Table Designer.

File|New|SQL Statement Query

Create a query for filtering data by selecting an existing SQL statement from among those built by using the Visual Query Builder.

File|New|Query

Open the Visual Query Builder to construct complex SQL statements for filtering data.

File|New|Form

Create a new form for viewing and entering data, using either the Form Expert or the Form Designer.

File|New|Home Page Form

Create a special form to serve as a home page by using the Home Page Expert.

File|New|Custom Form Class

Create a custom form class to use as a template for creating other forms with consistent recurring elements.

File|New|Report

Create a report based on table data by using either the Report Export or Report Designer.

File|New|Script

Create a new script (JavaScript program).

File|Open

Display the Open File dialog box that lets you browse directories to find IntraBuilder files. You can select file name, file type, and files within a database connection, if any. You can also choose to open the file in either Run or Design mode.

File|Save

Save the current file opened in Design mode.

File|Save As

Display the Save File window that lets you choose a file type, a file name, and browse for a folder in which to save the file.

File|Save As HTML

Display a Save As Dialog allowing you to save the current report as an HTML file for viewing (but not editing) on a web browser. You can name and specify a folder for the static HTML file.

Table|Save Row

Save changes to a row (record) after making modifications to a table in Run mode.

Table|Abandon Row

Abandon any unsaved changes you have made to a row (record). This ensures that if you move to another record, the unwanted modifications will not be automatically saved.

File|Close

Close the currently open file or the window or tool in focus.

File|Print

Display the Print dialog box to print the current file, and set options for print range and number of copies.

File|Database Administration

Access to security and referential integrity setup for different database types.

File|most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu..

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Most recently used files

A previously opened file. If the file is still in the same location it was in when you last closed it, you can open it again by clicking its name in the File menu.

File|Exit

Close the IntraBuilder Designer application.

Edit|Clear All Results

Delete contents of the Script Editor’s Results pane.

Edit|Insert from File

Display the Insert From File dialog box, which allows you to browse directories to find a JavaScript file containing code you wish to insert at the cursor location.

Edit|Copy

Copy the selected item to the Windows buffer. Use Edit|Paste to insert the copied item into a new cursor location.

Edit|Select All

Select all items or the entire data contents in the current window.

Edit|Copy to File

Display the Copy To File dialog box, which allows you to browse directories to find a JavaScript file to which you wish to currently selected code in the Script Editor.

Edit|Open File

Open the file specified by selecting a file name in a script displayed in the Script Editor.

Edit|Search|Find Text

Display the Find Text dialog box which allows you to search for a text string, optionally matching whole words, matching case, or searching up or down.

Edit|Search|Find Next Text

After searching for a text string by using the Find Text dialog box and closing it, you can repeatedly search for this text string by using the Find Next Text command or by pressing Shift F5.

Edit|Search|Replace Text

Display the Replace Text dialog box which allows you to search for a text string (optionally matching whole words, matching case, or searching up or down)-and replace it with another text string (optionally replacing all instances of the search text string).

Edit|Search|Go to Line Number

Display the Go To Line dialog box. Click the spin box or enter a line number and click OK. The cursor appears at the beginning of the line of code you specified.

Edit|Search|Top Line

Place the cursor at the beginning of the first line in the editor.

Edit|Search|Bottom Line

Place the cursor at the end of the last line in the editor.

Edit|Search|Find Ending Delimiter

After placing the cursor under an opening delimiter (such as a left brace), moves the cursor to the closing delimiter (right brace). Take care to place the cursor at the delimiter location, not to select it.

Edit|Search|Find Starting Delimiter

After placing the cursor under a closing delimiter (such as a right brace), moves the cursor to the opening delimiter (left brace). Take care to place the cursor at the delimiter location, not to select it.

Edit|Convert|to Uppercase

Convert selected text to all uppercase.

Edit|Convert|to Lowercase

Convert selected text to all lowercase.

Edit|Convert|to Initial Capitals

Convert selected text to initial capitals, that is, capitalizes the first letter of each word separated by a space.

Edit|Convert|to DOS text

Convert selected text from the Windows to the DOS character set.

Edit|Convert|to Windows text

Convert selected text from the DOS to the Windows character set.

Edit|Record Keystrokes

Record every subsequent keystroke. To record:

Choose Record Keystrokes and begin typing what you wish to record.

1. When you have finished typing the string you wish to record, reselect this command (or press F7) to stop recording.
2. Press F8 to insert the recorded string at the cursor location.

Edit|Playback Keystrokes

Insert, at the cursor location, the text string recorded by using Record Keystrokes (F7).

Edit|Comment Line(s)

Convert the current line or lines to a comment, placing double slashes in front of each line so that it will not execute. The current line is the line in which the cursor is located, or one or more highlighted lines.

Edit|Uncomment Line(s)

Convert the current comment line or lines to executable, removing the double slashes from in front of each selected comment line so that it will execute. The current line is the comment line in which the cursor is located, or one or more highlighted comment lines.

Edit|Execute Selection

In a script file, execute the currently selected section of code.

View|Toolbars and Palettes

Display the Toolbars and Palettes dialog box, which lets you set preferences, such as “image only” or “text only” or “both” for all IntraBuilder palettes and toolbars.

View|Explorer

Display the IntraBuilder Explorer which displays tables, forms, reports, home pages, scripts, images, and custom files under separate tab headings.

View|Script Pad

Display the Script Pad, in which you can enter JavaScript statements and expressions and view their results in a Results pane.

Script|Run

Run the current script in the Script Editor.

Script|Compile

Compile the current script in the Script Editor.

Table|Find Rows

When a form is open in Run mode, display the Find Rows dialog box. You select a field to search and type a value to find in that field, then press return to display the record.

Table|Replace Rows

When a form is open in Run mode, display the Replace Rows dialog box. In the Find What box you type a value to find in the Located in Field box. In the Replace With box, type the new replacement value. You can search for records with a certain value in one field, and add a value to the found records in another field.

Table|Begin Query By Form

To find a record in a running table by entering a value in any field, first choose Table|Begin Query by Form. The form's fields are cleared. Enter a value in any field. Choose Table|Apply Query by Form. The matching record is displayed.

Table|Begin Filter By Form

To display a view of a particular subset of records in a running table, such as January sales, choose Table|Begin Filter by Form. The form's fields are cleared. Enter a value in any field. Choose Table| Apply Filter by Form. The set of matching records (rowset) is selected; as you browse the table, you will see only matching records.

Table|Clear Filter By Form

In a running table, clear the selected rowset resulting from Table|Apply Filter by Form so that you can browse the entire table.

Structure|Manage Indexes

In Table Designer, this menu option asks you to save the current table structure. You cannot manage indexes until you have saved the current table structure.

Then, this menu option displays the Manage Indexes dialog box that lets you add new index keys and edit existing ones in the Define Index dialog box.

Table|Delete Rows

Display the Delete Rows dialog box which allows you to delete the current row, a specified row, or all rows (the entire table).Displays the Delete Rows dialog box which allows you to delete the current row, a specified row, or all rows (the entire table).

Structure|Pack Rows

Pack a table in which rows have been marked for deletion. When you delete a row (record) in a DBF table, the row no longer appears in Run mode, but it still occupies space in the sequence of rows; it has merely been marked for deletion but could still be recovered by using the vendor's database software. When you are the sole user of the table (that is, in Design Mode) you can pack the table; which completely deletes the marked rows, moving all the other rows up. In a large table, packing could take some time to execute.

Table|Sort Rows

Display the Sort Records dialog box. You can sort the rows (records) of the current rowset or table and save the sorted rows to a new table. Choose from a list of the current table's fields to sort the records according the selected fields, in either ascending or descending order. You can also exclude records where a field has a certain value.

Table|Count Rows

Display the Count Rows dialog box to count the number of records in the current table. Enter a record exclusion statement if you wish, then click OK. A window then appears with the record (row) count.

Table|Calculate Aggregates

Display the Calculate Aggregates dialog box which lets you calculate the average, minimum, maximum, or sum values for specified fields in the current table. You can add a SQL WHERE statement to filter the rows of the current table, that is to create a more limited rowset for this calculation.

Properties|Desktop Properties

Customize settings for IntraBuilder desktop environment, including current directory, search path, preferred external editor, and prompt preferences.

Properties|Script Pad Properties

Display the Script Pad Properties dialog box that lets you set font, text attributes, foreground and background color, editing preferences, and font preference for the Results pane.

Properties|IntraBuilder Explorer Properties

Customize properties for the IntraBuilder Explorer, including the options to show extensions and use supplemental search path.

Properties|File Item Properties

Customize properties for the selected file item in IntraBuilder Explorer, including file name, path name, file type, last changed, size, and so on.

Properties|Editor Properties

Display Editor Properties dialog box. Here you set preference for how all IntraBuilder editors (including Text Editor, Script Editor, Method Editor, and Script Pad) display fonts and format and highlight text elements.

Properties|Table Designer Properties

In Table Designer, display the Table Designer Properties dialog box, which lets you show or hide vertical or horizontal grid lines in the Table Designer window.

Properties|Form Designer Properties

Display the Form Designer Properties dialog box which lets you show the grid, enable snap-to-grid, show the ruler, and precisely set the grid dimensions.

Properties|Editor Properties

Display Editor Properties dialog box. Here you set preference for how all IntraBuilder editors (including Text Editor, Script Editor, Method Editor, and Script Pad) display fonts and format and highlight text elements.

Properties|Report Designer Properties

Display the Report Designer Properties dialog box, allowing you to show the ruler and the grid lines, and to specify that just one row or all rows be displayed on the report.

Window|Cascade

Rearrange all open windows on the Desktop in overlapping layers. The top line of each window is displayed so you can see the name.

If a window is minimized, this command has no effect on it.

Shortcuts

Keyboard: Shift+F5

Window|Tile Vertically

Display all open windows on the IntraBuilder desktop without overlapping them. When possible, the windows are sized equally on the screen.

Depending on how many windows you have open and how large your Desktop is, Tile Vertical attempts to place the windows next to each other. If you have more windows open than will fit vertically, Tile Vertical functions the same as Tile Horizontal.

If a window is minimized, this command has no effect on it.

Shortcuts

Keyboard: Shift+F4

Window|Tile Horizontally

Display all open windows on the IntraBuilder desktop without overlapping them. When possible, the windows are sized equally on the screen.

Depending on how many windows you have open and how large your desktop is, Tile Horizontal attempts to place the windows on top of each other. If you have more windows open than will fit horizontally, Tile Horizontal functions the same as Tile Vertical.

If a window is minimized, this command has no effect on it.

Window|Arrange Icons

Align the icons from minimized windows at the bottom of the IntraBuilder desktop.

This command has no effect on open windows.

Window|Close All

Close or minimize all the windows on the IntraBuilder desktop, including the IntraBuilder Explorer and Script Pad (if either is open).

When windows are closed, their names are removed from the open window list (Window menu) and added to the most recently used file list (File menu).

If you had open tables, this command does not close the files associated with those windows.

Window|Arrange Designer Windows

Display the Designer window and Inspector side by side at the top of the desktop, with Explorer minimized at the bottom of the desktop.

Help|Context Sensitive Help

Press F1 to display quick help on any highlighted or in-focus part of IntraBuilder. Context-specific help identifies menu commands and dialog boxes, briefly explaining their function. For details of usage, choose Help|Help Topics to see the Windows Help version of the IntraBuilder Developer's Guide.

Help|Help Topics

Open the IntraBuilder onscreen Help system.

Help|Views and Tools

Open the onscreen IntraBuilder Developer's Guide, with explanations of the IntraBuilder user interface, windows, tools, and other components.

Help|Language

Open the onscreen IntraBuilder Language Reference, describing all JavaScript objects and code elements.

Help|Keyboard

Open the IntraBuilder onscreen Help system to a list of keyboard shortcuts.

Help|How to Use Help

Open onscreen Help providing general guidance to using the Windows Help system.

Help|About

Display the IntraBuilder Version ID window, indicating version, registration, and amount of memory, GDI, and users resources available.

View|Sort Icons|By Name

Sort IntraBuilder Explorer file icons by name.

View|Sort Icons|By Type and Extension

Sort IntraBuilder Explorer file icons by file type, then by file extension.

View|Sort Icons|By Size

Sort IntraBuilder Explorer file icons by size.

View|Sort Icons|By Date and Time

Sort IntraBuilder Explorer file icons by date, then time.

View|Status Bar

Toggle switch to display or hide the Status Bar at the bottom of the IntraBuilder environment. The Status Bar displays the current mode, current statistics, dimensions, and other context-specific information depending on the type of file in use or the current operation. It also displays a line of explanation for IntraBuilder commands and dialog boxes selected with the mouse.

View|Large Icons

Display large icons to represent file types in the IntraBuilder Explorer tabs.

View|Small Icons

Display small icons to represent file types in the IntraBuilder Explorer tabs.

View|Details

In the IntraBuilder Explorer tabs, display file types in a single column with additional size, date, and time information.

View|Reset Custom File Types

Reset the file types listed when the Custom tab is selected as the default

View|Refresh

Update the display of IntraBuilder Explorer file items to reflect recent changes written to your disk storage.

File|Set Up Custom Components

Displays the Set Up Custom Components dialog box that lets you select from a list of custom components files, containing controls and data access components that you have pre-built and configured for convenient reuse.

File|Save As Custom

Display the Save As Custom dialog box which allows you to save either the selected components in a custom component (CC) file, or the form itself as a custom form class to create a base form as a template for future reuse. You provide a CC file name if saving components or a class name if saving a custom form class.

File|Set Custom Form Class

Display the Set Custom Form Class dialog box which allows you to set the current form to a custom form class. You can type the file name of the file containing the class or you can type the class. A browse button displays a directory navigation dialog box to help you find the file.

Edit|Select Form

Select the background form itself, that is, the current instance of the form class. Useful for shifting focus from selected components to the form itself.

View|Form

Run the current form displayed in Design mode. In Run mode you can test all the operations of the form on a linked table.

Same as toolbar button: 

View|Design Form

Shift the current form displayed in Run mode to Design mode.

Same as toolbar button: 

View|Inspector

Display the Inspector window for examining and editing the properties, events, and methods of Form Designer objects.

View|Method Editor

Display the Method Editor, which allows you to individually view and edit methods of the form's underlying JavaScript.

View|Go to Form Page Number

In multipage forms, display the Go To Form Page dialog box. Click the spin box or type the page number you want to see and click OK. That page is displayed.

View|Previous Form Page

In multipage forms, display the previous page of the form.

Same as toolbar button: 

View|Next Form Page

In multipage forms, display the next page of the form.

Same as toolbar button: 

View|Component Palette

When checked, display the Component Palette of standard controls and data access objects in the toolbar.

View|Field Palette

When checked, display the Field Palette of “live” or active fields linked to corresponding columns in the table or rowset specified by the form’s Query object.

Layout|Align|Left

Align the selected object in the Form Designer to the left side of the visual design surface.

Same as toolbar button: 

Layout|Align|Right

Align the selected object in the Form Designer to the right side of the visual design surface.

Same as toolbar button: 

Layout|Align|Top

Align the selected object in the Form Designer to the top side of the visual design surface.

Same as toolbar button: 

Layout|Align|Bottom

Align the selected object in the Form Designer to the bottom side of the visual design surface.

Same as toolbar button: 

Layout|Align|Absolute Horizontal Center

Align the selected object in the Form Designer to the middle of the horizontal axis of the visual design surface, that is exactly between the sides.

Layout|Align|Relative Horizontal Center

Align one or more selected objects in the Form Designer to the middle of the horizontal axis of area defined by the group selection, that is, equidistant from the sides of the selection area.

Layout|Align|Absolute Vertical Center

Align the selected object in the Form Designer to the middle of the vertical axis of the visual design surface, that is exactly between the top and bottom.

Layout|Align|Relative Vertical Center

Align one or more selected objects in the Form Designer to the middle of the vertical axis of area defined by the group selection, that is, equidistant from the top and bottom of the selection area.

Layout|Size|Grow to Largest Width

When selecting multiple form components of different widths, widen the smaller selected components to equal the width of the largest. This is useful for making many Text controls a consistent width.

Layout|Size|Grow to Smallest Width

When selecting multiple form components of different widths, narrow the larger selected components to equal the width of the smallest. This is useful for making many Text controls a consistent width.

Layout|Size|Grow to Largest Height

When selecting multiple form components of different heights, vertically enlarge the smaller selected components to equal the height of the largest. This is useful for making many *TextArea* and other components conform to a consistent height.

Layout|Size|Grow to Smallest Height

When selecting multiple form components of different heights, vertically shrink the larger selected components to equal the height of the shortest. This is useful for making many *TextArea* and other components conform to a consistent height.

Layout|Add Groups and Summaries

Display the Add Groups and Summaries dialog box. The Goups tab lets you choose the fields (columns) that will determine how to group the records (rows) in the report, and sort them in ascending or descending order (such as the Country column, grouping rows of each country together). The Summaries tab lets you choose the type of summary information the report will display for each group (such as a total of a numeric sales column).

Layout|Set Scheme

Display the Set Scheme dialog box which lets you choose title and label fonts, and background and foreground colors, save them as a scheme, and choose other saved schemes for your forms. You can visually unify a family of forms by using the same scheme for them.

Edit|Select Report

Select the background report itself, that is, the report class. Useful for changing focus from selected components to the report class.

View|Report

Run the current report displayed in Report Designer.

View|Report Design

Display the current running report in Report Designer.

View|Inspector

Display the IntraBuilder Inspector for viewing and editing the properties, events, and methods of report objects.

View|Method Editor

Display the Method Editor, which allows you to individually view and edit methods of the report's underlying JavaScript.

View|Group Pane

Toggle to show or hide the Group View of the Report Designer.

IDM_VIEW_REPORT_ZOOM_NORMAL

Display report at normal size. Roswell.

IDM_VIEW_REPORT_ZOOM_ENLARGED

Display report at large size. Enlargement enhances viewing only; does not change the appearance of the form when printed or deployed to browsers. Roswell

IDM_VIEW_REPORT_ZOOM_REDUCED

Display report at reduced size. Reduction alters viewing only; does not change the appearance of the form when printed or deployed to browsers. Roswell

View|Component Palette

Display the Component Palette, from which you can drag-and-drop standard controls and data access objects.

The palette appears embedded in the toolbar. You can select the entire palette (by clicking its outer right edge) and drag it out of the toolbar. The Component Pallete then becomes a floating palette that is always on top and in view.

View|Field Palette

Display the Field Palette, from which you can drag-and-drop pre-configured controls and data access objects. Convenient for reusing sets of custom controls linked to tables, rowsets, or databases.

View|Query Results

Display the selected query file in a form, showing a row resulting from the Query's SQL statements. The alternative is to choose View|Query Design to display the query's SQL statements in the SQL Statement Query Editor.

Structure|Add Field

In the Table Designer, add a field (column) to the current table structure, at the bottom of the list of fields.

Structure|Insert Field

In the Table Designer, insert a field (column) above the highlighted current field (cursor location) of the current table structure.

Structure|Delete Current Field

In the Table Designer, delete the highlighted current field (cursor location) of the current table structure.

Structure|Go to Field Number

In the Table Designer, display a dialog box with a Field Number spinbox that allows you to quickly move the cursor to the specified field (column) of the current table structure. Useful when editing large table structures.

Structure|Define Primary Key

In Table Designer (and for tables such as Paradox, that support primary keys), display the Define Primary Key dialog box. You select from a list of available fields for the current table, the fields of the primary key which will be used to generate the primary index. These fields must be consecutive fields, starting with the first field in the table structure.

View|Table

Switch current table (in Table Designer) into Run mode, so that you can browse and edit rows.

View|Query Design

Display the selected query file's SQL statements in the SQL Statement Query Editor. The alternative is to choose View|Query Results to run the query, showing a row resulting from the query's SQL statements in a form.

View|Table Design

Switch current running table into Design mode, so that you can view and redesign its structure.

Table|Edit Row

Make the current row (record) displayed in the running form modifiable. You can then edit the data in the record's fields, even if there is no Edit button on the form.

The Edit Row menu choice is affected by the *autoEdit* property of the *rowSet* of a Query object on the form. *autoEdit* must be set to *false* for Edit Row to be available.

Table|Add Row

In Run mode, display a blank form in which you can enter new data and save as a new row (record).

Table|Previous Row

Display the immediately preceding row (record) in the table. Same as the back arrow navigational button in the toolbar.

Table|Next Row

Display the immediately following row (record) in the table. Same as the forward arrow navigational button in the toolbar.

Table|First Row

Display the first row (record) in the table. Same as the beginning arrow navigational button in the toolbar.

Table|Last Row

Display the last row (record) in the table. Same as the end arrow navigational button in the toolbar.

View|Refresh Rows

After saving changes to rows in a running form, Refresh Rows updates the data displayed in the form from the table.

Report|Next Page

Display the next page of the current report in Run mode.

Report|Top Page

Display the first page of the current report in Run mode.

Report|Last Page

Display the last page of the current report in Run mode.

File|Import Image

When the Image Viewer (or an Image file) is open, display the Choose Image dialog box which allows you to find and select an image file to import into the IntraBuilder Image Viewer.

Properties|Image Viewer Properties

For an open Image file, display the Image Viewer Properties dialog box which allows you to size the window to the image or cause an animated GIF to play continuously.

File|Export Image

When the Image Viewer (or an Image file) is open, display the Export File dialog box which allows you to find and select an image file to export from the IntraBuilder Image Viewer.

Method|New Method

Display the Method Editor with a default template for a JavaScript function, to help you quickly write a new method.

1 Select the name “Method” and replace it with the name of the new method you wish to create.

2 Add your JavaScript statements between the braces and after the Export comment.

Note: The Export comment exports the JavaScript to the client for client-side execution; simply delete the statement if you want the method to execute on the server.

Method|Remove Method

When the Method Editor is open, remove the entire method code from the script.

Method|Verify Method

When the Method Editor is open, compile the current method and displays results in the Compilation Status window: current file name, number of lines, number of executable lines, number of errors, number of routines.

Method|Edit Event

Display the Edit Event dialog box, containing two lists. Select an object from the left, scrolling list, and then select one of that object's events in the right list. The selected event then appears in the Method Editor, ready for editing.

Method|Link Event

Display the Link Event dialog box, containing two lists.

1. Select an object from the left, scrolling list.
2. Select one of that object's events in the right list and click OK. The selected event is then linked to the object.

Events that are already linked are indicated by a little yellow lightning bolt icon.

Method|Unlink Event

Display the Unlink Events dialog box which shows list of the linked events for the current method or object shown in the Method Editor. Select the event you wish to unlink and click OK.

File|New Table

Create a new table by using either the Table Expert or the Table Designer. Conveniently available at the top level of the File menu when the New Table icon is selected in the Tables tab of the IntraBuilder Explorer.

File|New Query Builder Query

Create a new SQL query to filter table data by using the Visual Query Designer. Conveniently available at the top level of the File menu when the New Query icon is selected in the Queries tab of the IntraBuilder Explorer.

File|New SQL Statement Query

Display the New SQL Query Statement Editor in which you can type and save SQL statements to filter table data.

File|New Form

Create a new form for viewing and entering data, using either the Form Expert or the Form Designer. Conveniently available at the top level of the File menu when the New Form icon is selected in the Forms tab of the IntraBuilder Explorer.

File|New Custom Form Class

Open Form Designer on a new custom form class, when the new custom form class icon is selected in IntraBuilder Explorer's Forms tab:

The form design elements and components you create for a custom form class are saved as a non-running template (with the JCF extension) that can be used as a base form. Forms based on this custom form class inherit its contents.

File|New Home Page Form

Create a new home page for viewing and entering data, by using the Home Page Expert. Conveniently available at the top level of the File menu when the New Form icon is selected in the Forms tab of the IntraBuilder Explorer.

File|New Report

Create a new report for viewing or printing table data, by using either the Report Expert or the Report Designer. Conveniently available at the top level of the File menu when the New Report icon is selected in the Reports tab of the IntraBuilder Explorer.

File|New Script

Create a new script. Displays the Script Editor for composing JavaScript code. Conveniently available at the top level of the File menu when the New Script icon is selected in the Scripts tab of the IntraBuilder Explorer.

New Image

Display Microsoft Paint to create a new image. Conveniently available at the top level of the File menu when the New Image icon is selected in the Images tab of the IntraBuilder Explorer.

Delete

Delete any item selected in IntraBuilder Explorer, when right-clicking the item to display the short-cut menu.

Edit and View Table Rows

Run the current table selected in IntraBuilder Explorer's Tables tab when you right-click on the item to display the shortcut menu.

You can use the browse buttons in the toolbar to view the rows (records) and you can edit them.

Design Table

Open, in Table Designer, the current table selected in IntraBuilder Explorer's Tables tab when you right-click on the item to display the shortcut menu.

You can then modify the table's row and column structure.

File|Run Query

Display the selected query file in a form, showing a row resulting from the query's SQL statements.

File|Design Query

Display the selected query file's SQL statements in the SQL Statement Query Editor.

Run Form

Run the form selected in IntraBuilder Explorer’s Forms tab when you right-click on the item to display the shortcut menu.

Design Form

Open, in Form Designer, the form selected in IntraBuilder Explorer's Forms tab when you right-click on the item to display the shortcut menu.

Run Report

Run the report selected in IntraBuilder Explorer's Reports tab when you right-click on the item to display the shortcut menu.

Design Report

Open, in Report Designer, the report selected in IntraBuilder Explorer's Reports tab when you right-click on the item to display the shortcut menu.

Run Script

Run the script selected in IntraBuilder Explorer's Scripts tab when you right-click on the item to display the shortcut menu.

Compile

Compile the current form, table, report, custom components file, or other script, when you right-click on the item to display the shortcut menu.

An IntraBuilder file modified in the Script Editor or by writing JavaScript code should be compiled before running it or viewing it in a Designer.

Edit as Script

Open the current file (or table, form, custom form, report, custom component file, or other script right-clicked in IntraBuilder Explorer to display the shortcut menu) as JavaScript code in Script Editor.

Display Image

Display the image selected in IntraBuilder Explorer's Images tab when you right-click on the item to display the shortcut menu.

The image is displayed in the Image Viewer.

Design Image

Display the image selected in IntraBuilder Explorer's Images tab when you right-click on the item to display the shortcut menu.

The image is displayed in the image design program associated with the image file type in the Windows Registry. Standard BMP bitmap files are displayed in Microsoft Paint. Because JPEG and GIF image files are typically associated with HTML documents, these images may be displayed in your web browser. Modify the Windows Registry to associate specific image file types with the appropriate graphics design program.

Design Custom Form Class

Open for structural editing in Form Designer, the custom form class file (JCF) selected in IntraBuilder Explorer's Forms tab when you right-click on the item to display the shortcut menu. The custom form class create a base form that you use as a template to quickly create other forms with common elements. You cannot run a base form.

Run Executable File

Run any executable file selected in IntraBuilder Explorer's Custom tab when you right-click on the item to display the shortcut menu.

Works for EXE files and other program files like COM or BAT.

Open Application

Open any application’s file selected in IntraBuilder Explorer’s Custom tab, provided that the file type is registered in the Windows Registry to be opened by an available application.

Load Custom Components

Load the currently selected Custom Component file (CC) selected in IntraBuilder Explorer's Custom tab when you right-click on the item to display the shortcut menu.

Edit as Text

Display as text, the selected HTML file selected in IntraBuilder Explorer's Custom tab when you right-click on the item to display the shortcut menu.

Text appears in the Text Editor, which functions identically to the Script Editor and is subject to the same property settings as the other IntraBuilder editors.

Run HTML

Open in an HTML browser, the HTML file selected in IntraBuilder Explorer's Custom tab when you right-click on the item to display the shortcut menu.

Move Here

After right-click dragging a text selection, drops the selected text in the new cursor location, removing it from the original location.

Copy Here

After right-click dragging a text selection, copies the selected text in the new cursor location, leaving the selected text in place

Cancel

After right-click dragging a text selection, cancels the operation, leaving the selected text where it is.

Edit|Clear

Delete the selected item. Same as Delete.

Edit|Cut

Copy the selected item to the Windows buffer, removing it from its current location. Find a new location to paste the copied data.

Edit|Paste

Paste data copied or cut from another location in the new cursor location.

Edit|Undo

Reverse the last typing or clipboard operation. IntraBuilder stores as many as 32,767 characters in separate editing caches to allow multilevel undo. A session cache is emptied when you save the file or close the editing window. To save system resources, you can lower the cache limit in the Edit Properties dialog.

Undo reverses the following editing operations:

- Typing
- Edit|Cut
- Edit|Paste (you cannot Undo a record paste operation)
- Edit|Delete

The Undo command is not available if you have not performed an editing operation.

Shortcut

Keyboard: Ctrl+Z

Look In directory list

The Look In directory list is the current file search and save directory. If the Tables tab has focus, the list also shows all available database aliases.

To change the current directory, you can:

- Type a new drive and path directly into the directory input box;
- Click the down arrow to select from a list of previously visited directories (if any); or
- Click the folder button

 to open a directory selection dialog.

Note that the IntraBuilder Designer's current directory is not related to the current directory—or root directory—used by the IntraBuilder Server. Whether running on the same machine or on other machines on your network, the two components operate independently.

However, when you save forms, reports and other application components that are to run through your server, you must always keep the server's root structure in mind.

Note: Aliases are only listed in the Look In list when you have the Tables tab selected. They do not appear on the list when other file type tabs have the focus.

All IntraBuilder file types

IntraBuilder's All tab lists all the files, regardless of file type or extension, in the current directory.

Use the Explorer's Look In box to change directories or click the Folder button to browse. For help, click in the Look In box and press F1.

IntraBuilder's other tabs are filtered views of the current directory.

For information about the various file types shown in the All tab, click the other IntraBuilder Explorer tabs and press F1.

For more information about the selected form file, such as its full path name and date changed, right-click the form file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Forms

IntraBuilder's Forms tab lists the forms (identified by the JFM extension) and base forms (identified by the JCF extension) in the current directory.

To run a form file from the Forms tab, double-click the form file or right-click and choose Run Form from the shortcut menu.

To update a form, right-click the form file and choose Compile Form from the shortcut menu. The form file is updated to reflect the data in its linked tables or database.

To modify an existing form file in Form Designer, right-click the form file item and choose Design Form from the shortcut menu.

To create a new form, right-click the (Untitled) form file icon and choose New Form from the shortcut menu:

To create a new Custom Form Class, right-click the "open" (Untitled) form file icon and choose New Custom Form Class from the shortcut menu:

You use the Custom Form Class as a base form (template) for creating other forms that will share a common elements such as the same visual scheme, logo, layout and so on. You cannot a run a base form.

For more information about the selected form file, such as its full path name and date changed, right-click the form file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Reports

IntraBuilder's Reports tab lists the reports in the current directory, identified by the JRP extension.

To run a report file from the Reports tab, double-click the report file or right-click and choose Run Report from the shortcut menu.

To update a report, right-click the report file and choose Compile Report from the shortcut menu. The report file is updated to reflect the data in the linked tables or database.

To modify an existing report file in Report Designer, right-click the report file item and choose Design Report from the shortcut menu.

To create a new report, right-click the (Untitled) report file icon and choose New Report from the shortcut menu:

For more information about the selected report file, such as its full path name and date changed, right-click the report file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Scripts

IntraBuilder's Scripts tab lists the JavaScript files in the current directory, identified by the JS extension.

To run a script file from the Scripts tab, double-click the script file or right-click and choose Run Script from the shortcut menu.

To edit an existing script file in Script Editor, right-click the script file item and choose Design Script from the shortcut menu.

To create a new script, right-click the (Untitled) script file icon and choose New Script from the shortcut menu:

For more information about the selected script file, such as its full path name and date changed, right-click the script file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Tables

IntraBuilder's Tables tab lists table files in the current directory, as listed across the top of the tab. These file extensions are supported:

DB DBF

To run a table file from the Tables tab, double-click the table file or right-click and choose Edit and View Table Rows from the shortcut menu.

To modify the structure of an existing table file in Table Designer, right-click the table file item and choose Design Table from the shortcut menu.

To create a new table, right-click the (Untitled) table file icon and choose New Table from the shortcut menu:

For more information about the selected table file, such as its full path name and date changed, right-click the table file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Queries

IntraBuilder's Queries tab lists SQL query files from the current directory, as listed across the top of the tab. These file extensions are supported:

QRY SQL

To run a query file from the Queries tab, double-click the query file or right-click and choose Run Query from the shortcut menu. To open an existing query file in the Visual Query Builder, right-click the query file item and choose Design Query from the shortcut menu.

To create a new query, right-click the (Untitled) query file icon and choose New Query Builder Query or New SQL Statement Query:

For more information about the selected query file, such as its full path name and date changed, right-click the query file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Images

IntraBuilder's Images tab lists image files from the current directory, as listed across the top of the tab. These file extensions are supported:

GIF JPG JPEG BMP TIF TIFF XBM WMF EMF PCX EPS

To view an image file from the Images tab, double-click the file name or right-click the file icon and choose Display Image or Design Image from the shortcut menu.

To create a new image file, double-click the (Untitled) image file icon or right-click it and choose New Image from the shortcut menu:

Microsoft Paint or another registered graphics design program is opened.
For more information about the selected image file, such as its full path name and date changed, right-click the image file and choose Properties from the shortcut menu. Or choose Current File Item Properties from the main Properties menu.

Custom file types

IntraBuilder’s Custom tab lists files from the current directory, as filtered by the file specifications shown in the Custom Files input box.

To run or view a file from the Custom tab, double-click the file name or right-click the file icon and choose an action from the shortcut menu. Available actions may differ among file types. If, for example, you select an *.HTM file, you can load it into your browser or edit it as text.

For more information about the selected file, such as its full path name and date changed, choose Properties from an item's right-click shortcut menu or choose Current File Item Properties from the main Properties menu.

Custom Files input box

Change the file filter specification used to list files from the current directory.

To remove any file filter, highlight and then delete it.

To add a specification, type it into the input box. All specifications must be separated by commas.

To run or view a file from the Custom tab, double-click the file name or right-click the file icon and choose an action from the shortcut menu. Available actions may differ among file types. If, for example, you select an HTM file, you can load it into your browser or edit it as text.

For more information about the selected file, such as its full path name and date changed, choose Properties from an item's right-click shortcut menu or choose Current File Item Properties from the main Properties menu.

Term not found

Please search the IntraBuilder Help index for the word or phrase you want.

Term not found

Please search the IntraBuilder Help index for the word or phrase you want.

Term not found

Please search the IntraBuilder Help index for the word or phrase you want.

Term not found

Please search the IntraBuilder Help index for the word or phrase you want.

Term not found

Please search the IntraBuilder Help index for the word or phrase you want.

Term not found

Please search the IntraBuilder Help index for the word or phrase you want.

Method Editor

The Method Editor displays just the methods in the JavaScript code underlying IntraBuilder forms, components, and other objects. You can set its properties (choose Properties|Editor Properties) to help you view, write, and structure the methods and events that control IntraBuilder components.

(Note: The Method Editor is a powerful object-oriented programming tool that requires some familiarity with JavaScript. Be sure to read the onscreen documentation in IntraBuilder Help to gain an understanding of the concepts you need to know to work with this tool.)

The Inspector's Methods tab displays the current object's built-in methods, that is, the methods pre-defined for the component. You can call these methods in methods you create with the Method Editor. Methods you create in the Method Editor can also be inspected in the Methods tab.

If the current form or report already has methods, the first method in the Inspector's method list is current in the Method Editor window. If the form doesn't have methods, the "Header" section of the Method Editor is pre-selected.

To create a new method, click the Inspector's Events page, choose an event, then click the tool button to the right of the text box. The Method Editor opens automatically (or it becomes active if it's already open). A new method template is created and linked to the current event. You can then write statements for the new method. Or you can display the Edit Event dialog box to link the event to an existing method.

The Method menu offers commands to simplify writing methods in the Method Editor. The Event commands offer a dialog box as an alternative to using the Inspector to edit a selected objects events. You can also display these menu options (along with cut, copy, paste, and Method Editor properties) in a shortcut menu by right-clicking within the Method Editor.

Script Editor

The Script Editor displays all the JavaScript generated by creating IntraBuilder forms, reports, and home pages in the Experts and Designers.

To view or edit a script in the Script Editor, close the Form Designer, right-click on a file and choose Edit As Script from the shortcut menu.

Debugging. When you run an IntraBuilder application and IntraBuilder detects a problem, you are alerted. To fix the problem, click the Alert's Fix button; the Script Editor appears with the problem line indicated by the cursor position.

The Script Editor is highly customizable with many properties for color-coding and working with JavaScript.

Inspector, Object Selector

The Inspector displays the properties of components (both controls and data access objects), as well as their events and methods.

The name of the current object (or class) appears in the objector selector, the selection list box at the top of the Inspector. Click the Down arrow of the selector to display a drop down list of objects associated with the current form or report. Select an object; its properties, events, and methods appear in the Inspector tabs below.

Inspector, Item Heading

The properties, events, and methods listed in the Inspector tabs may appear preceded by a plus (+) or minus (-) sign. An item name with the plus prefix means that it is heading; you can open it by double-clicking which displays a list of its content immediately below. When an item heading is open (or expanded) the item heading is preceded by a minus (-) sign.

Form Designer Visual Design Surface

This is the visual design surface on which you position text, graphics, controls, and data access objects. The window size, grid dimensions, snap-to-grid behavior, and rulers are adjustable in the Form Designer Properties dialog box (choose Properties|Form Designer Properties).

In Form Designer you create forms visually, by selecting functional controls from the Component Palette (such as HTML, data entry fields, list boxes, buttons, and check boxes) that you click and position on the visual design surface. You can move object about on the surface, forcing them to automatically align to gridlines (if you prefer) and resizing them.

To quickly align objects, match their sizes, or set the overall color and font scheme, use the options in the Layout menu.

Table Designer, Field Name

Enter a name for the field (up to 10 characters for DBF; up to 25 for DB).

You can enter letters, numbers, and underscores, but no other characters. The first character must be a letter.

DB and most SQL tables allow spaces; DBF does not.

Table Designer, Field Type

Specify the field type by selecting the type you want from the list. Which type you select determines what kind of data the field will contain, and whether you can set the width, decimals, and index options for this field.

Table Designer, Field Width

Specify the field size. In the case of DBF files you can change field size for character, numeric, and float fields only (all others have fixed width).

Table Designer, Field Decimal

Specify the number of digits allowed to the right of the decimal point (for float and numeric fields only). In the case of DBF files, float and numeric fields have no decimals selected, by default.

You can set decimals to a maximum of 2 less than the width value you define. The total width including decimal settings, the decimal point, and an optional minus sign, must be 20 characters or less.

Table Designer, Field Index

Specify the index to determine whether to index records using the values in this field (for character, date, float, and numeric fields only).

Ascend	Index this field in ascending order (for character fields, this is ASCII order, or the order determined by your language driver).
Descend	Index this field in descending order.
None	Omit this field from indexing (or removes an existing index associated with this field). DEFAULT

If you select Ascend or Descend for a DBF table, the Table Designer creates an index for the field in the multiple index file (.MDX) associated with the table.

Table Designer, Table Type

Specify the type of table you want to create. You can always create standard DB (Paradox) and DBF (dBASE) tables. These types are standard for Borland Database Engine. If you select Paradox (the DB type), a Field Properties Inspector appears on the IntraBuilder desktop.

Other database types (such as Access MDB or various SQL databases) may be available if you have configured them at the server with the Borland Database Engine Configuration Utility. (Check the level of client/server support offered by your edition of IntraBuilder.)

Set Active Database

When you select a table type other than the standard (DB and DBF) types, this dialog box appears to let you select a database of the selected type. It shows the available databases if you have configured them at the server with the Borland Database Engine Configuration Utility.

Database Administration dialog box

Set security passwords and referential integrity rules for the specified table types.

Table Type	Choose a table type.
Security	Click to display the Administrator Password dialog box (for DBF tables) or the Security dialog box (for DB tables).
Referential Integrity	Click to display the Referential Integrity dialog box, for those table types (like DB) that support it.

To open this dialog box, choose File|Database Administration while in the Table Designer or running a form.

Administrator Password dialog box

Enter your administrator password to access security for DBF type tables. Asterisks appear in the text-entry field as you type. Click OK. The Security dialog box will appear so you can set access levels for users and tables.

To open this dialog box, in Run mode choose File|Database Administration, and click the Security button.

Password dialog box

Enter your password to access security for DB type tables. Asterisks appear in the text-entry field as you type. Click OK.

To open this dialog box, in Run mode choose File|Database Administration, and click the Security button.

DBF Security dialog box (Users page)

Set access levels for users, and access-level privileges for DBF tables and fields within specific DBF tables.

Displays authorized users (and their group associations) for this table.

New—displays the New User dialog box where you authorize new users, giving them a user name, password, and access level.

Edit—displays the Edit User dialog box where you edit the user information for the user name currently selected in the Users page.

Delete—Delete the currently selected user name from the authorized users. The deleted user will no longer be able to open the current DBF table.

DBF Security dialog box (Table page)

Set access levels for users, and access-level privileges for DBF tables and fields within specific DBF tables.

1. Find and select a table for which you will set table- and field-level access privileges.
2. Select a table from the list.
3. Click the folder button if you need to navigate through directories to find the table you want. Only DBF tables are displayed.
4. When you click the Edit Table button, the Edit Table Privileges dialog box appears.

DBF Security dialog box (Enforcement page)

Set access levels for users, and access-level privileges for DBF tables and fields within specific DBF tables.

Choose one of two security enforcement schemes:

When Loading IntraBuilder	When a user attempts to load IntraBuilder itself, a login is required, thus preventing unauthorized users from meddling in your intranet system.
When Opening an Encrypted Table	Whenever a user tries to view a form linked to an encrypted DBF table, a login is required. Thus anyone may use unencrypted tables, but unauthorized users are prevented from accessing protected tables

New/Edit User dialog box

Authorize a new user to access the DBF tables, or edit the user security information for a currently authorized user of DBF tables.

After you have entered all these settings, click OK to save the new user information.

User	Enter the user name, according to your system conventions.
Group	Enter the user's group name, such as "Marketing".
Password	Enter the user's password.
Access level	Enter a number from one to eight. Lower values provide greater access.

Edit Table Privileges dialog box

For the selected DBF table, define various table-access and field-access privileges for up to 8 access-levels for each user group.

Group	Assign the current table to a group. A DBF table can be assigned to only one group. The group name is matched with a user group name to enable access. Click on the down arrow to display a list of the available groups from the Group list (you created these groups when you added users in the New User dialog box).
Table Access Levels	For each type of table operation, specify the most restricted access level that can perform that operation: Read—View the table contents Update—Edit existing records in the table Extend—Add records to the table Delete—Delete records from the table To set table privileges, select a value (1–8) for each operation. Lower access numbers indicate the greatest access; higher numbers indicate the greatest restriction.
Field Privileges	For each field, set one of three access privileges granted to each of 8 access levels. Full—View and modify the field (default) Read Only—Only view the field; no editing allowed. None—No access. The user access level with this privilege prevents the user from seeing the field altogether; the field does not even appear on the form.

Security (DB) dialog box

To set security passwords for encrypted DB (Paradox) tables, first select an existing DB table in the Table list box, then click Edit Table.

Directory	Click the folder button if you need to navigate through directories to find the table you want. Only DB tables are displayed.
Edit Table	Displays the Edit Table Privileges dialog box.

To open this dialog box, choose File|Database, set the Table Type to Paradox, and click the Security button.

Master Password dialog box

Set the master password required to access an encrypted DB (Paradox) table.

Password	Type the password required to access the selected DB table. Only asterisks appear as you type.
Confirm	Retype the password exactly as you did in the first text-entry field. Only asterisks appear as you type.

Referential Integrity Rules dialog box

Add, edit, or delete referential integrity rules for the currently selected DB (Paradox) table. To use this dialog box you must first either login to a database or, in IntraBuilder Explorer, select a directory containing tables (such as DB) that support referential integrity.

Referential integrity means that a field or group of fields in one table (the “child” table) must refer to the key of another table (the “parent” table). Only values that exist in the parent table’s key are valid values for the specified field(s) of the child table.

New	Displays New Referential Integrity Rule for constructing relationships between fields of related tables.
Edit	Displays Edit Referential Integrity Rule for modifying existing relationships between fields of related tables.
Drop	Remove the highlighted referential integrity rule.

To open this dialog box, choose File|Database Administration, set the Table Type to Paradox, and click the Referential Integrity button.

New/Edit Referential Integrity Rule dialog box

For the selected database login or directory of tables, establish the relationship between parent and child tables. From the available child fields, specify a child field related to the primary key field.

Rule Name	The name of the referential integrity rule between a parent and child table. The default name is a concatenation of the names of the parent and child tables.
Parent Table	Choose a parent table from the drop-down list.
Child Table	Choose a child table from the drop-down list.
References	Relate child fields to primary key fields.
Update Behavior	Choose Restrict or Cascade, if supported by the table type.
Delete Behavior	Choose Restrict or Cascade, if supported by the table type.
Relationship	Choose One-to-One or One-to-Many.

To open this dialog box, choose File|Database Administration, set the Table Type to Paradox, and click the Referential Integrity button. Then click New or Edit.

Toolbars and Palettes dialog box

Set preferences for the display of toolbars and palettes in the IntraBuilder environment.

Toolbars and Palettes	Check the toolbars and palettes you want to appear by default whenever you open Form Designer or Report Designer.
Component Palettes	Choose how IntraBuilder will identify components on the Component Palette: Image Only—just the icon appears Text Only—no icon, just the component name Image and text—both icon and name appear
Field Palettes	Choose how IntraBuilder will identify linked components on the Field Palette: Text Only—no icon, just the field name Image and text—both an icon and field name appear
Show Tabs	When checked, palettes are organized into tabbed pages. The Component Palette displays two pages for Controls and Data Access Objects. The Field Palette displays a separate page for the fields linked through each active Query object.
Mouse Revert to Pointer	When working in Design mode, uncheck the Revert To Pointer option to make the pointer remain a control or field icon after you place the object. This lets you place multiple instances of an object without having to return to the Component or Field Palette to select the object each time. You can change the pointer back to its default behavior by clicking the Pointer control on the Component or Field Palettes.
Large buttons	Enlarges the buttons on the toolbar. The Palette icons are not affected.
Tooltips	Display brief text explanations of each icon or button as the pointer is passed over it. Tooltips are helpful as you are learning; uncheck Tooltips to turn it off when you no longer need it.

To open this dialog box, choose View|Toolbars And Palettes.

Edit Event dialog box

Choose an event to edit from the events associated with the available objects. When you click OK the selected event appears in the Method Editor.

Objects	The objects currently available on the form
Events	The events associated with the currently selected object

This dialog box presents all the objects on the report and their events. Select an object from the Object pane on the left and that object's events are listed in the Event pane on the right. Notice that some objects don't have any associated events. Select an object and one of its events and the Method Editor will open with the selected event in its current form. You can edit the event to perform specific operations.

To open this dialog box, choose Method|Edit Event from the Form or Report designer.

Link Event dialog box

Set up a link between the method currently in the Method Editor and an event.

Objects	The objects currently available on the form
Events	The events associated with the currently selected object

This dialog box establishes links between events and methods. If you want a given event to trigger a method, you can link that method to multiple events. The Link Event dialog box displays the name of the method currently displayed in the Method Editor, all the objects on the report in a pane on the left, and each object's events in a pane on the right. While editing a method in the Method Editor, select Link Event from the Method Menu, then select an object in the left pane, and an event in the right pane, and the method will be linked to that event.

To open this dialog box, in the Method Editor choose Method|Link Event.

Unlink Events dialog box

Choose an event to unlink from the events associated with the available objects.

Events to Unlink The events associated with the currently selected object

This dialog box breaks a link between a specific method and specified events. The Unlink Events dialog box displays the method currently in the Method Editor at the top of the box and the events linked to that method in the pane below. Select the events that you want to dissociated from the method and click the OK button.

To open this dialog box, in the Method Editor choose Mehtod|Unlink Event.

Manage Indexes dialog box

After saving the current table structure, add new index keys and edit existing ones in the Define Index dialog box.

Index Name	Group for index fields.
Index Key	The field used for sorting.

To open this dialog box, in the Table Designer choose Structure|Manage Indexes.

Define Index dialog box

Create a new index.

Options	Sort order options based on table type.
Available Fields	Ascending Descending Include Unique Key Values Only Include Duplicate Keys
Fields of Index Key	The fields in the table. An index key is a set of indexes. You can select a group of indexes and give the group a name. That name is called an Index Key.

To open this dialog box, in the Table Designer choose Structure|Manage Indexes, then press the New button.

Define Primary Key dialog box

Select from a list of available fields of the current table, the fields that will be used as the primary key. A primary key is a set of indexes that will be used to sort the records in the table.

The fields of a primary key must be consecutive fields, starting with the first field in the table structure.

Available Fields	The fields in the table.
Fields of Index Key	An index key is a set of indexes. You can select a group of indexes and give the group a name. That name is called an Index Key.

To open this dialog box, in the Table Designer choose Structure|Define Primary Key.

New Report dialog box

Click Expert to enter the first of seven Report Expert dialog boxes. Click Designer to go to the Report Designer.

If you aren't sure whether to enter the Expert or the Designer, go to the Expert. The Expert is a step-by-step procedure that will familiarize you with the options and choices available to you when building a report. After running the Expert you will have a usable report. Then, if you want to, you can make changes to that existing report in the Designer.

New Table dialog box

Click Expert to enter the first of two Table Expert dialog boxes.

Click Designer to go to the Table Designer.

If you aren't sure whether to enter the Expert or the Designer, go to the Expert. The Expert is a step-by-step procedure that will familiarize you with the options and choices available to you when building a table. After running the Expert you will have a usable table. Then, if you want to, you can make changes to that existing table in the Designer.

Table Expert, Step 1 of 2

Use an existing table as a model for the new table.

To build your new table using fields previously defined for other tables:

1. Choose a table from the list of Sample Tables (at the left). The fields in that table are displayed in the box in the center labeled From Sample Table.
2. Click one of the fields to select it.
3. Click the right arrow button. The selected field moves over to the box on the right labeled For New Table.

Use any or all of the sample tables to build the new table. You are not limited to a single sample table. However, several sample tables have fields with a name used in another table. Your new table cannot have more than one field with a specific name (that is, field names must be unique within a given table).

Table Expert, Step 2 of 2

Choose the format of the new table. There are several choices available in the list box labeled Table Type.

This step also allows you to either run the new table or enter the Table Designer. The Table Designer presents you with the flexibility to change any aspect of the new table. You can change any of the things that you built using the Table Expert.

For information on how to use the Table Designer refer to the onscreen Help while in the Designer. Click Designer and then go to the Help menu.

Click Run to see what the new table looks like. Once in Run mode, you can always return to Design mode.

New Form dialog box

Click Expert to enter the first of six Form Expert dialog boxes.

Click Designer to go to the Form Designer.

If you aren't sure whether to enter the Expert or the Designer, go to the Expert. The Expert is a step-by-step procedure that will familiarize you with the options and choices available to you when building a form. After running the Expert you will have a usable form. Then, if you want to, you can make changes to that existing form in the Designer.

To open this dialog box, choose File|New|Form.

Form Expert, Step 1 of 6

Specify a table or query file to use as the basic data source for the new form.

The top box, labeled Look In, allows you to browse directories to find the tables that you want. When you get to a directory that contains tables, those tables are displayed in the lower box, labeled Selected Table or Query File.

Click on the table you want to use and press the Next button (or double-click the table).

Form Expert, Step 2 of 6

Select the fields that you want to use from the source table or query.

The fields in the data source (table or query) are displayed on the left in the box labeled Available. Click a field to select it and press the right arrow button. That field moves to the box on the right labeled Selected.

You need to use at least one field, but you don't have to use all of them. If you want to copy all of them, press the double right arrow button and all the fields move to the Selected box.

When you have selected all the fields that you want to include on the new form, press the Next button.

Form Expert, Step 3 of 6

Specify the layout style for the new form.

Columnar Layout	Specifies that the fields in a record be displayed in a column, that is, one field on a line.
Form Layout	Specifies that fields be run on to a single line where possible.

Form Expert, Step 4 of 6

Select a display scheme for the form.

Sample	In the upper left, the Sample box depicts the current scheme. This box shows examples of the Title, the text, and the background used on the form. You can change all these things separately or select a prebuilt scheme using the Scheme listbox.
Title tab	Select a font and color to be used for the title of the form.
Label tab	Select a font and color to be used for the text on the form (including the links to other pages)
Form tab	Select a color to be used as a background. With the Background tool, you can specify a background graphic to be used on the form. If the graphic is smaller than the form, the background graphic will be tiled on the resulting form. There are several prebuilt graphics available from the Background Image listbox that make pleasant backgrounds for a form. If you want to use a different graphic image, click the Background Image tool to display the Choose Image dialog box. Browse to the directory that contains the image that you want and click Open.
Save Scheme	After you decide on the elements of your scheme you can save that scheme for use on other forms. Press the Save Scheme button, and give the scheme a name. That new name will appear in the Scheme listbox

You can change all these attributes later (in the Form Designer), so don't worry about making a mistake. When you are satisfied with the display scheme, press the Next button.

Form Expert, Step 5 of 6

Step 5 allows you to do three different things:

- Choose between button controls and image controls to perform row operations;
- Select which row operation buttons are displayed on the form;
- Specify links to other forms and reports.

There are two choices for the appearance of the controls:

Buttons	The buttons have text on them that give the name of their respective operations.
Images	The images are icons that represent the operation graphically.

There are three types of row operations: Navigate, Update, and Search or Limit.

Navigate	Display the various records of the data source: First—displays the first record Last—displays the last record Next—displays the next record in sequence Previous—displays the previous record in sequence
Update	Make changes to the data source: Add—adds a new record to the source table Delete—deletes the currently displayed record from the source table Edit—enables you to make changes to the currently displayed record Save—saves the changes made to the currently displayed record Abandon—ignores the changes made to the currently displayed record
Search or Limit	**Query by Form** The New Query button allows you to query the table for a record that contains a specific value in a specific field. For example, if you want to find anyone named Smith, you can press New Query, type Smith in the last name field, then press Run Query. The next record with last name of Smith is displayed. **Filter by Form** The New Filter button allows you to specify a filter condition by typing the condition into any of the fields on the form. For example, if you want all the people named Smith, press New Filter, type Smith in the last name field, then press Run Filter. The first record with last name of Smith is displayed.

To place the controls on the new form, click the checkbox next to the name of each of the operations, or press the All button at the lower right of the Row Operations box.

To link from the new form to another form or another report:

1. Use the two boxes in the Links to Other Objects area (near the bottom of the window) to specify a form and a report that you want to reference from the new form.
2. Click the tool button next to the Form box; the Open File dialog box appears.
3. Navigate to a directory that contains a form that you want to reference. Do the same for a report. Controls to run the linked form and report will be displayed on the new form.

If you selected button controls, the form and report control will read Run Form and Run Report. If you selected image controls, an icon for the form and the report will be displayed instead of a button.

Form Expert, Step 6 of 6

Decide to run the new form or open the Form Designer to modify it. The Form Designer presents you with the flexibility to change any aspect of your new form. You can change any of the things that you built using the Form Expert. You can also run the form after entering design mode.

For information on how to use the Form Designer, refer to the onscreen Help while in the Form Designer. Click Designer and then go to the Help menu.

Click Run to see what the new form looks like. You can go into the Designer from Run mode.

New Home Page Form dialog box

Click Expert to enter the first of four Home Page Form Expert dialog boxes.

Click Designer to go to the Form Designer (which is the same Designer used to build Home Pages).

If you aren't sure whether to enter the Expert or the Designer, go to the Expert. The Expert is a step-by-step procedure that will familiarize you with the options and choices available to you when building a home page. After running the Expert you will have a usable home page. Then, if you want to, you can make changes to that existing home page in the Designer.

Keep in mind that a Home Page is a type of form. You will use the Form Designer to make changes to the home page that you build with the Home Page Expert.

Home Page Form Expert, Step 1 of 4

The Home Page Expert assumes you are building a Home Page for your company. Step 1 of 4 of the Home Page Expert gives you the opportunity to build a home page that has a title (the Company Name), a graphic image (the Company Logo), a text field for a slogan or motto (the Company Description), and a text field for an e-mail address.

All of these things can be changed later (in the Form Designer), so don't worry about making a mistake. Go ahead and put something on the home page so that you will have a starting point for future modifications.

To add the Company Name, type the name in the text box labeled Company Name.

To add the Company Logo, identify an existing image by clicking on the folder icon on the right of the box labeled Company Logo. Clicking on the folder icon displays the Choose Image dialog box, which allows you to browse through directories to locate the image file that you want to use as the company logo. After locating an image you can place the logo by checking one of the buttons in the Logo Placement area.

To add a company description, type the text that you want to appear on the home page into the box labeled Company Description. This text could be a company motto or slogan, or any other text you want to appear on the home page.

To add an e-mail address to the home page, type the address in the box labeled E-Mail Address.

Home Page Form Expert, Step 2 of 4

To display links on your home page that connect to other pages on your network site:

1 Click the folder icon to the right of the box labeled Directory. This displays the Choose Directory dialog box, which allows you to specify a directory that contains the forms and reports that you want to access from the home page. Select a directory.

2 All the forms and reports in the selected directory appear in the box labeled Available Forms and Reports. Click one of them and press the right arrow button. That file name moves to the box labeled Selected Links. If you made a mistake, you can move the file name back to the Available Forms and Reports box by clicking the left arrow button.

3 Click on a file name in the Selected Links box. Under the Selected Links box is the Description box. The Description box is used to construct the link that will appear on the home page. On the home page, the link will look like text and that text is specified in the Description box. Type in the title or name of the form or report.

Home Page Expert, Step 3 of 4

Select a display scheme for the home page.

Sample	In the upper left, the Sample box depicts the current scheme. This box shows examples of the Title, the text, and the background used on the home page. You can change all these things separately or select a prebuilt scheme using the Scheme listbox.
Title tab	Select a font and color to be used for the title of the home page.
Label tab	Select a font and color to be used for the text on the home page (including the links to other pages)
Form tab	Select a color to be used as a background. With the Background tool, you can specify a background graphic to be used on the home page. If the graphic is smaller than the home page, the background graphic will be tiled on the resulting home page. There are several prebuilt graphics available from the Background Image listbox that make pleasant backgrounds for a home page. If you want to use a different graphic image, click the Background Image tool to display the Choose Image dialog box. Browse to the directory that contains the image that you want and click Open.
Save Scheme	After you decide on the elements of your scheme you can save that scheme for use on other forms. Press the Save Scheme button, and give the scheme a name. That new name will appear in the Scheme listbox

You can change all these attributes later (in the Form Designer), so don’t worry about making a mistake. When you are satisfied with the display scheme, press the Next button.

Home Page Expert, Step 4 of 4

Step 4 allows you to either run the new home page or enter the Form Designer. The Form Designer presents you with the flexibility to change any aspect of your new home page. You can change any of the things that you built using the Home Page Expert.

For information on how to use the Form Designer, refer to the onscreen Help while in the Form Designer. Click Designer and then go to the Help menu.

Click Run to see what the new home page looks like. You can go into the Designer from Run mode.

File Item Properties dialog box

Displays file information about the file item currently selected in IntraBuilder Explorer.

Name	Name of the currently selected file.
Path	Path to the file (directory location)
Type	Type of the file (such as “table”)
Rows	Number of rows (records) in file
Last Changed	Date the file was last modified
Size	Number of bytes in the file
(Not) Read Only	Whether the file is locked from editing
(Not) Archived	Whether an archive backup has been made

To open this dialog box, select a file in the IntraBuilder Explorer and choose Properties|Current File Item Properties.

IntraBuilder Explorer Properties dialog box

Choose options for behavior of IntraBuilder Explorer:

Show Extensions	Shows file type extensions in IntraBuilder Explorer.
Use Supplemental Search Path	If checked, includes files in the IntraBuilder search path.

To open this dialog box, choose Properties|IntraBuilder Explorer Properties.

Desktop Properties dialog box, Files page

Customize settings for IntraBuilder desktop environment, including current directory, search path, preferred external editor, and prompt preferences.

Current Directory	Click the folder button to display a Choose Directory dialog box. This sets the default current directory for the IntraBuilder Explorer.
Search Path	Click the folder button to display a Choose Directory dialog box. This sets the default search path for the IntraBuilder Explorer.
Backup files	If checked, displays backup files in IntraBuilder Explorer.
External Editor File Name	Click the tool button to display the Choose Script Editor dialog box. This sets the default text or program editor that is invoked when you open a script for editing.

To open this dialog box, choose Properties|Desktop Properties.

Desktop Properties dialog box, Application page

Customize settings for IntraBuilder desktop environment, including current directory, search path, preferred external editor, and prompt preferences.

Prompt for Experts	When you double-click a new (Untitled) table, form, or report icon, a dialog box appears prompting you to choose either the Expert or the Designer. By unchecking these boxes you can skip this dialog, causing the new table, form, or report to open in Design mode.
Remember Logins	Check this so that you don't have to type in a login string for frequently used databases.
Display System Tables	Uncheck this to hide system tables, which are distracting if it is not your job to maintain them.
Multiuser/Lock	When checked, locks tables you have open so that others cannot modify them at the same time.
Inspector Outline	When checked (the default) the Inspector displays properties by category. Uncheck to make the Inspector display properties alphabetically.
Splash Screen	Uncheck this to skip the display of the IntraBuilder logo at startup.
MRU List Size	Click the spinbox to change the number of Most Recently Used files that can be displayed in the File menu.

To open this dialog box, choose Properties|Desktop Properties.

Table Designer Properties dialog box

Set display preferences for Table Designer, to show or hide grid lines.

Horizontal grid lines	If checked, Table Designer separates rows with horizontal lines.
Vertical grid lines	If checked, Table Designer separates columns with vertical lines.

To open this dialog box, from the Table Designer choose Properties|Table Designer.

Find Rows dialog box

Find a specific row (record) in the current rowset or table by specifying a value for one of the columns (fields) of the target row.

Located in Field	Select one of the rowset's columns.
Find What	Type the value you expect to find in that field of the target row.
Search Rules	Click a radio button to choose either: Partial Length — The search string will be found within larger strings. Exact Length—The search string must match exactly (such as a whole word). Match Case—The case of the search string must be matched.

To open this dialog box, choose Table|Find Rows.

Replace Rows dialog box

Find rows with specified data content in particular fields and replace those values with a new value.

Find What	Type the value you expect to find in that field of the target row.
Located in Field (left)	Select one of the rowset's columns in which you expect to find the value you specify in the Find What box.
Replace With	Enter the new value or data content you want to place in the found row's field you specify in the right Located in Field box.
Located in Field (right)	Select one of the rowset's columns in which you want to place the new value you typed in the Replace With box.
Search Rules	Click a radio button to choose either: Partial Length—The search string will be found within larger strings. Exact Length—The search string must match exactly (such as a whole word). Match Case—The case of the search string must be matched.

To open this dialog box, choose Table|Replace Rows.

Delete Rows dialog box

Delete rows (records) of the current table or rowset. The radio buttons offer three options:

Current	Deletes only the current row (displayed on the form).
Specified	Deletes the row specified by the SQL statement you type in the box.
All	Deletes all rows in the current table or rowset.

To open this dialog box, choose Table|Delete Rows.

Sort Rows dialog box

Sort rows (records) in the current rowset or table and save the sorted rows into a new table.

Available Fields	Select from a list of the current table's fields, the first key field by which you want the rows to be sorted.
Order By	The key field you click appears in the Order By box. You can add more than one field to the Order By box, in cases where you may have more than one row with the same value in the key field. After grouping rows by the first key field, rows within that grouping are ordered by the second key field, and so on.
Select Key Field Sort Direction	Choose to sort the records according the selected fields, in either ascending or descending order.
Limit Rows	Enter a SQL WHERE statement to limit the row selection. You can thereby exclude records where a field has a certain value.

To open this dialog box, choose Table|Sort Rows.

Count Rows dialog box

Counts the number of rows (records) in the current rowset or table and displays the number in a dialog box.

Click OK to the count all the rows in the current table. A dialog box then appears with the row count.

To limit the count to a rowset within the table, enter a SQL WHERE statement in the **Where** box, then click OK.

To open this dialog box, choose Table|Count Rows.

Calculate Aggregates dialog box

Calculate the average, minimum, maximum, or sum values for specified numeric fields (columns) in the current table.

In the **Calculation** area, click a radio button to indicate:

Average	Average of the numeric values in all the fields of the specified rowset.
Minimum	Minimum value found in all the fields of the specified rowset.
Maximum	Maximum value found in all the fields of the specified rowset.
Sum	Sum total of the numeric values in all the fields of the specified rowset.

In the **Available Fields** area, choose a field of the current table for calculation. Only those numeric fields that *can* be calculated are shown.

In the **Limit Rows** area, enter SQL WHERE statement in the Where box to filter the rows of the current table, that is to create a more limited rowset for this calculation.

To open this dialog box, choose Table|Calculate Aggregates.

Params Property Builder dialog box

Create and manage name-value pairs in an array for the *params* property of a Java applet.

To create a name-value property:

1. Type a name in the Name box
2. Type its associated value in the Value box.
3. Click Add. The new name-value pair appears in the list below. Repeat these steps to add additional name-value pairs to the array.

To remove a name-value pair from the array:

1. Select the pair in the Parameters list.
2. Click Remove.

To open this dialog box, in the Inspector click the params property tool.

Report Designer Properties dialog box

This dialog box allows you to set various properties of the Report Designer.

Visual Aids	Show Grid—displays the sections of the Report View when checked.
	Show Ruler—displays the ruler at the margins of the Report View when checked.
Rowset Display	One Row—displays only one row from the source table.
	All Rows—displays all source table rows that will fit on the report page.

To open this dialog box, from the Report Designer choose Properties|Report Designer.

Script Pad Properties dialog box, Results Page

Set position of Results Pane and font preferences for the Results Pane.

Position	Choose one of four positions in which the Results pane will appear in the Script Pad window: Top, Bottom, Left, or Right.
Font	Shows the current font selected for the Results Pane. To choose a new typeface, type style, and typesize, click the adjacent tool button to display the Font Property Builder.
Reset	Revert Font setting to the default: Courier 9.

To open this dialog box, from the Script Pad choose Properties|Script Pad Properties.

Editor Properties dialog box, Editor Page

Customize settings for IntraBuilder text editors, including cursor, tab, spacing, layout, fonts, colors and other display and editing properties. These settings affect all four editors: Script Pad, Method Editor, Script Editor, and Text Editor.

Set editing preferences and formatting behavior by selecting the checkboxes:

Editor Speed Setting	Use this dropdown listbox to switch base editors. Choices are IntraBuilder and BRIEF. There are minor differences—some keyboard shortcut mappings, for example—but in most major respects the two editors offer similar functionality.
Reset	Revert editor preferences to default settings.
Auto Indent	When you press Enter, the new line is indented to match the indent in the previous line. If Auto Indent is off (unchecked), the cursor moves to the left margin when you press Enter.
Backspace Outdents	If checked, lets you skip over indenting by pressing the Backspace key.
Optimal Fill	If checked, converts groups of spaces to tabs when you load a file. If unchecked, spaces are preserved.
Use Tab Character	Toggles the use of tab and the equivalent number of spaces.
Cursor Through Tabs	Determines what happens when an arrow key is pressed at a tab mark. If checked, the arrow key take you through the tab one character at a time. If unchecked, the arrow keys move the insertion point across the whole tab. When you press the right or left arrow keys and you're at the beginning or end of a tab, this setting determines whether the arrow key moves you through the tab space by space or whether you skip to the end of the tab.
Smart Tab	Determines whether the tab key positions the insertion point to the starting column position of the previous line when you press tab and you are to the left of that point.
Brief Cursor Shapes	Toggle between the BRIEF style (horizontal) and IntraBuilder (vertical) cursor shapes.
Group Undo	Determines whether all preceding editor commands and actions of the same type that have been executed since the last time Enter was pressed are “undone” when you choose Edit|Undo. If Group Undo is false, then only the last keystroke or command is undone.
Undo After Save	Normally, when a file is saved, the undo cache is cleared. That is, any edits you made before the save cannot be undone. This option lets you override that behavior, allowing you to undo your most recent actions even after saving a file.
Persistent Blocks	If columnar mode is on and this option is checked, a highlighted block of text remains highlighted until you select a new block. If the option unchecked, or if columnar mode is off, a highlighted block is automatically deselected when you click an area outside of the block . As noted, the Persistent Blocks option is only available when columnar

	mode is on. To toggle columnar mode on or off, press Alt+C.
Overwrite Blocks	Replaces a marked block of text with whatever is typed. If Persistent Blocks is also on, then typed text is added rather than substituted for the marked text.
Cursor Beyond EOF	If checked, lets you place your cursor anywhere on the page. If unchecked, the cursor cannot be placed beyond the last entered line.
Use Syntax Highlight	Determines whether syntax highlighting and formatting settings are applied to files with a DBF source file extension. Untitled files edited in the Method Editor and text typed into the Script Pad all assume syntax highlighting when this option is checked. Existing files with non-DBF source extensions do not use syntax highlighting.
Visible Right Margin	Adds a vertical line in the editor window to mark the position of the right margin. To change the position of the marker, use the Right Margin spinbox Default is 80 points from the left.
Interpret Text As	Choose DOS or Windows text conventions. (Not applicable in Script Pad)
Mouse Speed	Drag the pointer to increase or decrease the speed with which the pointer moves over text.
Line Length	Specify the maximum line length in the text and program editors as well as the Script Pad. The setting is applied to new edit windows or a reopened Script Pad; it does not apply to the current window or any open editing windows). If you type beyond this line length or paste data into the window that contains any line that exceeds the maximum, an error message appears.
Tab Size	Specify the tab width. This setting is applied immediately to all edit windows, including the Script Pad.
Block Indent	Specify the indent of code blocks.
Undo Limit	Specify maximum number of bytes in the temporary cache that contains all current data available for Undo operations in any edit window.

To open this dialog box, from the Method Editor choose Properties|Editor Properties.

Editor Properties dialog box, Font Page

Customize settings for IntraBuilder text editors, including cursor, tab, spacing, layout, fonts, colors and other display and editing properties. These settings affect all four editors: Script Pad, Method Editor, Script Editor, and Text Editor.

Choose a typeface and type size for the default font used in IntraBuilder editors. Changes are displayed in the Sample area at the middle of the dialog box.

Name	Dropdown list shows all monospaced typefaces available on your system.
Size	Dropdown list shows all available sizes for each available typeface.
Reset	Restore the default typeface and typesize.

To open this dialog box, from the Method Editor choose Properties|Editor Properties.

Editor Properties dialog box, Appearance Page

Customize settings for IntraBuilder text editors, including cursor, tab, spacing, layout, fonts, colors and other display and editing properties. These settings affect all four editors: Script Pad, Method Editor, Script Editor, and Text Editor.

Set color and text attributes for each type of code element for easy readability and faster editing. Changes are displayed in a sample viewer at the bottom of the dialog box.

Appearance Speed Setting	Dropdown list from which you can choose a preset color scheme. Selecting a scheme here overrides any custom settings you may have selected.
Reset	Restore all default color settings for the currently selected Appearance Speed Setting.
Elements	A scrolling list of code element types. Select a code element, then choose colors and text attributes for that element
Color	A palette from which you can apply color to a selected text element. First choose an element, then left-click for a foreground color and right-click for a background color.
System Color	Click these check boxes ON to use the default foreground or background colors for text elements.
Text Attributes	Chose Bold, Italic, or Underline for the selected code element.
Sample	Shows how the different code elements will appear in text editors, reflecting the settings on this page.

To open this dialog box, from Method Editor choose Properties|Editor Properties.

PaperSize Property Builder dialog box

Select one of seven different paper sizes as the size of paper to be used to print the report.

Most printers can use letter size (8.5 x 11 inch) and many can use legal size (8.5 x 14 inch). If your printer can print on envelopes or on labels, then some of the other selections on the listbox will be useful.

The PaperSize property itself is a number from 1 to 7. The property value itself corresponds to the following list of paper sizes:

Letter

Legal

Executive

A4

Comm #10 Envelope

Monarch Envelope

DL Envelope

To open this dialog box, from the Report Designer select the top-level object and click the Printer property tool, then the Papersize property tool.

PaperSource Property Builder dialog box

You can select the source of the paper that your printer will use (for a printer with more than one method of loading paper). The PaperSource property varies depending on the type of printer installed. Typically the PaperSource property is a number from 1 to 15.

The property value corresponds to a paper source. For different printers, the PaperSource Property Builder will display the number and corresponding paper source. For example, the PaperSource Property Builder will display the following list for a typical printer:

AutoSelect Tray	Lets the printer decide which tray to load
Upper	Loads paper from the upper tray on printers with two trays
Lower	Loads paper from the lower tray on printers with two trays
Envelope	Envelopes require a special loading mechanism
Manual Feed	Loads from the manual feed mechanism

To open this dialog box, from the Report Designer select the top-level object and click the Printer property tool, then the Papersource property tool.

Form Designer Properties dialog box

Set display preferences for Form Designer's visual design surface, to show or hide grid lines and rulers, and to set grid line widths.

Form settings	Grid and ruler options: Show Grid—Select to show grid lines. Snap to Grid—Select to make components align automatically to nearest vertical and horizontal grid lines. Show Ruler—Display both vertical and horizontal rulers.
Grid settings	Set the widths between both vertical and horizontal grid lines: Fine, Medium, or Coarse.
Custom	Allows you to independently set the exact widths for horizontal and vertical grid lines by using the spin boxes
X Grid	Set the exact width of the horizontal grid lines.
Y Grid	Set the exact width of the vertical grid lines.

To open this dialog box, from the Form Designer choose Properties|Form Designer Properties.

Options Property Builder dialog box

The Select object gives the user the ability to select items from a drop-down list. The Items in the list can be two different types: Array or Filename

1 If you select Array as the Type, the items in the list are taken from an array of elements that you build.
Select Array from the Type box, and click the tool icon to the right of the Data Source box to display the Build Array dialog box.

2 If you select Filename as the Type, the items in the drop-down list are the filenames of all the files in the current directory. This gives the user to ability to select files.
Select Filename in the Type box, and press OK. If you want to constrain the displayed filenames, you can the constraint in the Data Source box. For example, typing *.txt in the Data Source box constrains the displayed filenames to only those that have an extentsion of TXT.

Need more information on select list options?

Would you like more general information on the Select object?

To open this dialog box, in the Inspector click the options property tool (under Data Linkage Properties).

Build Array dialog box

Add elements to the array that is used by the selected Select object. You can add two types of elements to the array:Strings and Expressions.

Type a string in the String box and press the Add button and the new string will move to the box labeled Array Elements. The string box checks to make sure that the entered characters fit the definition of a string (almost any characters including spaces).

The Expression box checks that the characters entered fit the definition of a valid expression. A valid expression consists of an alphanumeric variable (like “x” or “Var1”) or a datatype (like “number”) followed by and equals sign (=) and then a value (a number) or an already defined variable (like a Boolean true).

Need more information on select list options?

Would you like more general information on the Select object?

To open this dialog box, in the Inspector click the options property tool (under Data Linkage Properties) to open the Options Property Builder. Set the Type to Array, and click the Data Source tool.

Template Property Builder dialog box, String page

Build a template that will constrain the characters allowed in the selected HTML control or text control.

Build the template in the box labeled Template and choose one of the two tabs: String and Numeric. Click the String tab to select string symbols from the box at the bottom. Click the Numeric tab to select numeric symbols.

To open this dialog box, in the Inspector click the template property tool (under Edit).

Template Property Builder dialog box, Numeric page

Build a template that will constrain the characters allowed in the selected HTML control.

Build the template in the box labeled Template and choose one of the two tabs: String and Numeric.

String tab Click on the String tab to select string symbols from the box at the bottom.

Numeric tab Click on the Numeric tab to select numeric symbols.

For example, if you wanted the text to be limited to 3 digits, a decimal, and 2 digits to the right of the decimal (nnn.nn):

1. Click the Numeric tab
2. Double-click 9 three times
3. Double-click the period once
4. Double-click the 99 twice more (999.99).

Characters typed in the HTML or text control will beconstrained such that typing 456789 will cause 456.78 to be displayed.

To open this dialog box, in the Inspector click the template property tool (under Edit).

SQL Property Builder dialog box

Inspect a Query object on a form, click the SQL property

SQL Statement	Click here and type a SQL statement or statements in the SQL Satement pane at the bottom of the dialog box.
SQL Statement File	Type a file name containing the desired SQL statements or click the tool button to display an Open dialog box for locating and opening files.
Query Builder File	Type a file name of a file created by using the Visual Query Builder or click the tool button to display an Open dialog box for locating and opening files.

To open this dialog box, in the Inspector click the sql property tool.

Text Property Builder dialog box

Apply HTML tags to the text of the currently selected text object on the form. You can apply HTML tags for color, font, links, and any other HTML formatting and functions to any text object, including create text labels for fields and form titles.

Type, in the upper right Text Without Tags pane, the text you want to be displayed by the form's currently selected text object. Then select the parts of that text to which you wish to apply HTML tags.

Font Tags	Apply HTML tags for font styles to the text selected (highlighted) in the Text Without Tags pane at the right: Bold Italic Underline Strikethrough Subscript Superscript
Text Without Tags	Shows the current, plain text of the currently selected text object. To apply HTML tags using the controls in this dialog box, you must select the portion of the text in this window to which you want the tags to apply. You can add more text in this window.
URL Tag	Type the name of a file or URL that you want to link to the selected text. Then click Add. The <A HREF...> tags appear around the selected text in the Text With Tags pane at the lower right.
Color Tag	To select a predefined HTML color from the drop-down list, click the arrow button. Then click add to apply the FONT COLOR tag to the text selected in the Text Without Tags pane. To create a new custom color, click the color tool button. The Color Property Builder appears, in which you can choose a new color and give it a custom name. The custom color will then appear in the drop-down list of custom colors.
Custom Tags	To select a predefined custom HTML style tag from the drop-down list, click the arrow button. Then click Add to apply the HTML tag to the text selected in the Text Without Tags pane. To create a new custom HTML tag, click the New button. The Manage Custom Tags dialog box appears so you can enter a new custom tag name and enter its start and end tags. To modify the custom HTML tag currently appearing in the selection box, click the Edit button. The Manage Custom Tags dialog box appears with the current custom tag's name and its start and end tags.
Tags At Current Position	This drop-down selection list shows all the tags that start at the current cursor position, so as the cursor is moved, the list of tags changes. If a block of text is selected, this list displays all the HTML tags that have been applied to the current text selection. To view a list of multiple tags applied to text, select the text in the Text Without Tags pane. You must select the entire section of text to which the tags are applied. Then click the Tags at Current Position arrow to display the list.

	To remove a tag from a text selection, click the arrow, select a tag from the drop-down list, then click the Remove button.
Text With Tags	This pane displays the tagged text as it might be interpreted by an HTML browser. It shows the results of the HTML tags you have applied in this dialog box.

To open this dialog box, in the Inspector click the text property tool.

Manage Custom Tags dialog box

Identify tags that are not available from the Text Property Builder dialog box or construct custom HTML tags that can combine several HTML formatting instructions.

Tag Description	Enter the new custom tag's name (that is, the name you give it). The name is the character string that will appear in the Custom Tags combobox on the Text Property Builder dialog box. You can use up to 50 characters and spaces (only the first 30 characters are shown in the Tag description combobox on the Manage Custom Tags dialog box.
Start Tag	Enter the HTML tag that you will use to start the custom formatting.
End Tag	Enter the HTML tag that you will use to end the custom formatting.

To open this dialog box, in the Inspector click the text property tool, then click the New or Edit button.

Save as Custom dialog box

IntraBuilder Explorer displays two (untitled) icons.

The “full” icon opens as a new form with the JFM extension.

The “empty” icon opens as a new custom Form class (with the JFC extension).

A custom form class, called a “base form” serves as a template (with standard elements, such as company logos, animated GIF files, links, and so on, preset and ready to go) for creating new forms with the same look-and-feel. (Note: You cannot run a base form.)

To create a new base form:

1. Use the Form Designer to create the common features of the form.
2. Choose File|Save as Custom to display the Save as Custom dialog box.
3. Choose Save Form as Custom, then complete the rest of the dialog box.

Save Selected Components as Custom	If you have designed custom components for use on the form, click this radiobutton.
Place in Component Palette	Click the Place in Component Palette checkbox to put the custom components on the Component Palette. The Component Palette will contain the custom controls when you use the template form.
Class Name	Type the name of the class. The name can be any character string.
Custom Component File Name	Type the name that you want to use for the file that contains the custom component. Custom component file names have an extension of CC. You can click the tool button and browse directories for existing CC files.

To open this dialog box, from the Form designer, choose File|Save As Custom.

Set Custom Form Class dialog box

The Set Custom Form Class dialog box is used to save the current form as a custom form class. In other words, the form you are currently designing will become the template for other forms.

To save the current form as a custom form class:

1 File Name Containing Class
Type the name of a form that you want to use as the template. You can click the tool button and browse directories for a form to use as the template.

2 Class Name
Type the name of the class. The name may be any character string.

To open this dialog box, from the Form Designer choose File|Set Custom Form Class.

Set Up Custom Components dialog box

To set up custom components,

1. In the Set Up Custom Components dialog box, click Add. The Choose Custom Component dialog box appears.
2. In the Choose Custom Component dialog box, choose the custom component file (with the CC extension) that you created for (or into which you saved) your custom component. Click Open.
3. The path name to the selected custom component file now appears in the Set Up Custom Components dialog box.
4. Click Add. The custom components you have saved in your CC file appear in a new Custom page of your Component Palette.

To open this dialog box, from the Form Designer choose File|Set Up Custom Components.

Set Scheme dialog box

Select or create a preset visual scheme for a form.

Sample box	Depicts the current scheme. This box shows examples of the Title, the text, and the background used on the form. You can change all these things separately or select a prebuilt scheme using the Scheme listbox.
Title tab	Select a font and color to be used for the title of the form.
Label tab	Select a font and color to be used for the text on the form (including the links to other pages).
Form tab	Select a color to be used as a background. With the Background tool, you can specify a background graphic to be used on the form. If the graphic is smaller than the form, the background graphic will be tiled on the resulting form. There are several prebuilt graphics available from the Background Image listbox that make pleasant backgrounds for a form. If you want to use a different graphic image, click on the Background Image tool to display the Choose Image dialog box. Browse to the directory that contains the image that you want and click Open.
Save Scheme	After you decide on the elements of your scheme you can save that scheme for use on other forms. Press the Save Scheme button, to access the Save Scheme dialog box and give the scheme a name. That new name will appear in the Scheme listbox.

You can change all these attributes whenever you wish. When you are satisfied with the display scheme, press the Next button.

To open this dialog box, from the Form or Report designers, choose Layout|Set Scheme.

Scheme Name dialog box

Provide a name to a set of font and color attributes for the form.

Type any character string up to 30 characters. You can use letters, numbers, and spaces.

Image Viewer Properties dialog box

Set the properties of the image viewer.

Size Window to Image	If checked, the image will be shown in a window that is the same size as the image. If not checked, the image will be contained in a standard size window. If the image is larger than the view window, scroll bars will appear. If you check Size Window to Image and then open an image, changing the size of the viewer window will distort the image.
Play Continuosly if Animated Image	If the image is an animated GIF image, the Image Viewer will play the image. If this radiobutton is checked, the image will play continuously while the image is open in theviewer. If this radiobutton is not checked, the image will play only once and stop on the last frame.

To open this dialog box, choose Properties|Image Viewer Properties.

IntraBuilder Login dialog box

Type the Group name, your user name and your password.

If you don’t know what these things are, contact you system administrator. These fields are not defined by IntraBuilder, but by the person in your group that is responsible for system and database security.

Find Text dialog box

Type the search string (any characters including spaces) in the Find What box.

If applicable, select Match Whole Words (to only locate whole words that match the search string) or Match Case (to only locate search strings that match case exactly).

The search by default will proceed down the file (towards the end). If you want to search up (towards the beginning of the file, click the Up button in the Direction box.

To open this dialog box, from an editor choose Edit|Search|Find Text.

Replace Text dialog box

Type the search string (any characters including spaces) in the Find What box.

Type into the Replace With box the text that you want to replace the search string.

If applicable, select Match Whole Words (to only locate whole words that match the search string) or Match Case (to only locate search strings that match case exactly).

The search by default will proceed down the file (towards the end). If you want to search up (towards the beginning of the file, click the Up button in the Direction box.

To open this dialog box, from an editor choose Edit|Search|Replace Text.

Font Property Builder dialog box

Changes you make in this dialog box are reflected only in IntraBuilder, except for font styles. Font styles are reflected in both IntraBuilder and browsers. Select the settings you want, and click OK.

If you want to make other font changes that will affect the look of a form on a user's browser, use the Text Property Builder. You can access the Text Property Builder through the Inspector: with a text object selected, click the tool icon to the right of the text property.

To open this dialog box, in the Inspector click the Font Properties, fontName tool.

Color Property Builder/Choose Color dialog box

The Color Property Builder (or Choose Color) dialog box allows you to choose a color for the selected object. There are several methods for selecting a color.

- Type the standard hexadecimal color code in the box at the top.
- Click one of the Basic Colors (in the box on the left) then click OK.
- Construct a custom color. To do that,
  1. Click in the color matrix (the multicolored box) on the right to pick the combination of Red, Green, and Blue.
  2. Move the crosshairs up and down to select the Saturation.
  3. Move the crosshairs left and right to select the Hue.
  4. The vertical bar on the right determines the Luminance (darkness) of the color by the position of the triangular pointer on the right of the bar. The result of your choices is displayed in the Color/Solid box.
  5. When you get the color you want, press the Add to Custom Colors button. The new color appears in one of the previously white boxes directly above the Add to Custom Colors button. Then click the OK button.
- Type the color component numbers. To do that:
  1. Type the numbers for Saturation, Hue, Luminance and Red/Green/Blue directly into the respective boxes. The result of your choices is displayed in the Color/Solid box.
  2. When you get the color you want, press the Add to Custom Colors button. The new color will appear in one of the previously white boxes directly above the Add to Custom Colors button. Then click the OK button.
- Locate the color you want in the color wheel:
  1. Put the crosshairs on the color you want.
  2. Adjust the Luminance (darkness) with the vertical bar to the right of the color matrix. The result of your choices is displayed in the Color/Solid box.
  3. When you get the color you want, press the Add to Custom Colors button. The new color will appear in one of the previously white boxes directly above the Add to Custom Colors button. Then click the OK button.

To open this dialog box, in the Inspector click the Visual Properties color tool.

DataSource Property Builder dialog box

Image objects are related to an image file by the object's data source property. The Location of the image file is either a file name or a binary. File name is the most common source. Only a simple object like a radio button can use a binary location. A radio button is either on or off, so a binary location can be any Boolean field in a table.

To relate a graphic image to an object,

1. Select Filename as the Location.
2. Click the tool button to the right of the Image box and the Choose Image dialog box will appear.
3. Navigate to the directory that contains the image that you want to relate to the selected image object and click the Open button.

To open this dialog box, in the Inspector click the dataSource property tool.

Open dialog box

Use this dialog box to locate and open a file.

To open a database you must have set up an alias to that database directory. Run the BDE configuration utility to set up a BDE alias.

Save dialog box

Use this dialog box to save your file in the current directory (or press the browse icon to navigate to another directory).

Print dialog box

Displays and lets you change settings for the current print job. To accept the settings and continue with the print job, press OK. To abort the print job, choose Cancel. For help on individual controls and features in the Print dialog, click your right mouse button on a control, then click the "What's This" pop-up item.

Query Parameter Property Builder dialog box

Use the Query Params Property Builder to build and maintain parameters used in queries.

The Parameter Name box lists all the parameters used with a specific query object. Click on a parameter name to display the parameter value's data type and value in the Parameter Value box.

To open this dialog box, in the Inspector click the params property tool.

Stored Procedure Parameter Property Builder dialog box

Use the Stored Procedure Params Property Builder to build and maintain the parameters used in stored procedures.

The Parameter Name box lists all the parameters used with a specific stored procedure object. Click on a parameter name to display the parameter's type (in the Parameter Type field) and the parameter value's data type and value (in the Parameter Value field).

The parameter can be one of four types: Input, Output, Input/Output, or Result.

To open this dialog box, in the Inspector click the params property tool.

Choose Field dialog box

Use the Choose Field dialog box to link a text object to a field in a query or table.

The Queries box lists all the queries and tables that are currently active on the form. Choose the query or table that contains the field that you want to link to the selected text object.

The Fields box lists all the fields in the query or table that is specified in the Queries box. Choose the field that you want to link to the selected text object.

Then press the OK button.

Remote agents

Installation and usage instructions for remote agents are covered in the file SERVER.HLP, located in your IntraBuilder root directory (default c:\program files\borland\intrabuilder).