www.realbasic.com

User's Guide

REALbasic 2006 User's Guide

Documentation by David Brandt.

Printed in U.S.A.

Mailing Address	REAL Software, Inc. 1705 South Capital of Texas Highway Suite 310 Austin, TX 78746
Web Site	http://www.realsoftware.com
ftp Site	ftp://ftp.realsoftware.com
Support	REALbasic Feedback at the REAL Software web site.
Bugs/ Feature Requests	Submit via REALbasic Feedback at the REAL Software web site.
Database Plug-ins	The REALbasic CD; the most recent versions are at www.realsoftware.com.
Sales	sales@realsoftware.com
Phone	512-328-REAL (7325)
Fax	512-328-7372

Version 2006R1, January, 2006

Contents

CHAPTER 5 **Programming with Events and Objects** 217

CHAPTER 1

Introduction

Before you get started developing applications with REALbasic, there are a few things you should know. Reading this chapter will help you understand how to install REALbasic and how to get answers to your questions.

Contents

Welcome to REALbasic

REALbasic makes it easy to build powerful applications quickly. If you are new to programming, you will find REALbasic's programming language easy to learn. If you are an experienced programmer, you will find the language to be powerful. In either case, you will find you can accomplish quite a bit in a short period of time.

REALbasic has a visual graphical user interface ("GUI") builder that lets you build your application's user interface without any (or very little) programming. If you know how to drag and drop, you can build an interface. REALbasic provides a rich set of interface controls and you can create your own controls as well.

REALbasic's programming language is an object-oriented version of the BASIC programming language. BASIC is an acronym that stands for Beginners All-Purpose Symbolic Instruction Code. It was originally designed to be used for teaching programming. Consequently, its syntax is less cryptic and easier to understand than most languages. REALbasic supports most of BASIC's commands. However, that is where the similarities between BASIC and REALbasic end.

Most forms of BASIC are interpreted. This means that they include a translator that has to constantly translate BASIC code into the code that the computer can actually understand. REALbasic has no interpreter. REALbasic compiles your code when you run your application.

REALbasic's form of the BASIC language is also "object-oriented." This means that it uses a modern architecture that most popular programming languages (like C++ and Java) are using today. Object-oriented programming languages make it easier to write and debug because the code is written as individual objects that are similar to objects in the real world. In fact, in many ways REALbasic is more object-oriented than languages like C++ and certainly easier to learn and program.

REALbasic also makes application development faster and easier than traditional languages by removing the need to learn how to access the programming interface for the operating system. This application programming interface (or "API" for short) consists of thousands of commands, not one of which you ever need to learn to build applications in REALbasic.

Installing REALbasic

The REALbasic application has the following hardware and operating system requirements:

Windows Requirements

To run REALbasic on Windows, you must have the following:

- A PC with at least a 1.0GHz processor and at least 256MB of RAM (1.5GHz processor, 512MB of RAM recommended),
- The Windows 98, ME, NT, 2000, or XP operating system.

Run the Windows installer to install REALbasic 2005 for Windows.

Linux Requirements

To run the REALbasic IDE on Linux, you must have the following:

- A PC with at least a 1.0GHz processor and at least 256MB RAM; 512MB and 1.5GHz processor recommended,
- Any x86-based Linux distribution that includes GTK+ 2.0 (or higher), Glibc-2.3 or higher, and CUPS (Common Unix Printing System). This includes Novel Linux SuSE 8.1 or higher, Red Hat Enterprise Linux 3 or higher, Mandrake 8.1 or higher.

Macintosh Requirements

To run REALbasic on Macintosh you must have the following:

- A Macintosh with a 600MHz G3 processor that runs Mac OS X 10.2.8 or above; 800GHz G4 or better, Mac OS X 10.3.9 or above recommended.
- A hard disk with at least 120 megabytes of free space available to install REALbasic and at least 512MB RAM; 768MB RAM recommended.

  To install the Macintosh version of REALbasic from the CD ROM, drag the REALbasic folder from the CD-ROM to your hard disk. If you downloaded it from the REAL Software web site, double-click the disk image on your desktop and then drag the REALbasic 2005 folder to your hard disk. It is recommended that you store this folder in your Applications folder.

Where to Begin

If you are new to programming, you should begin by going through the *QuickStart* and then the *Tutorial*. This will give you a good overview of REALbasic and introduce you to the programming language. Next, read the *User's Guide*. This guide will provide you with detailed information on the language and the various components that make up REALbasic. When you need details about a specific control or command in the language, consult the *Language Reference*.

Documentation Conventions

This documentation uses the following typographical conventions:

Initial References

The first time a new phrase or term is used, it will appear in *italics* for *emphasis*.

Menu References

When you are told to select a menu item, the menu name is listed first, followed by an arrow, then the item name and command key shortcut. For example File ▶ Quit means "choose Quit from the File menu".

Keyboard Equivalents

Most menu items have keyboard equivalents. On Windows and Linux, the Ctrl key is the primary modifier, and sometimes the Alt and Shift keys are used. On Macintosh, the Command key is the primary modifier; sometimes the Option and Shift keys are also used. When keyboard equivalents are given, Windows and Linux equivalents are given first, followed by the Macintosh keyboard equivalent. For example "Ctrl+Q or ⌘-Q" means "Ctrl+Q on Windows and Linux or Command-Q on Macintosh."

Screen Illustrations

REALbasic is truly a cross-platform application. The application itself runs under Windows 98 and above, Linux with GTK 2.x installed, and Mac OS X. REALbasic can build applications that run on Windows 98/ME, Windows NT/2000, Windows XP Home/XP Professional, Mac OS X, Mac OS 8-9 (a.k.a. "classic"), and Linux. The screen snapshots of the REALbasic development environment are a mixture of Windows XP, the Linux "Bluecurve" desktop, and Mac OS X. Where necessary, windows and controls that appear in built applications are shown for the Windows, Macintosh, and Linux platforms. Some interface features of REALbasic are specific to a platform, so only that platform is shown.

Code Examples

Code examples appear in gray boxes:

```
Dim i, x as Integer
x=0
For i = 1 to 100
 x = x + i
Next
```

Icons

There are four icons used to call your attention to steps and important notes:

This icon means that there are numbered steps for you to follow.

This icon means that the text to the right of it is supplemental information that clarifies a point or is relevant only to some REALbasic users.

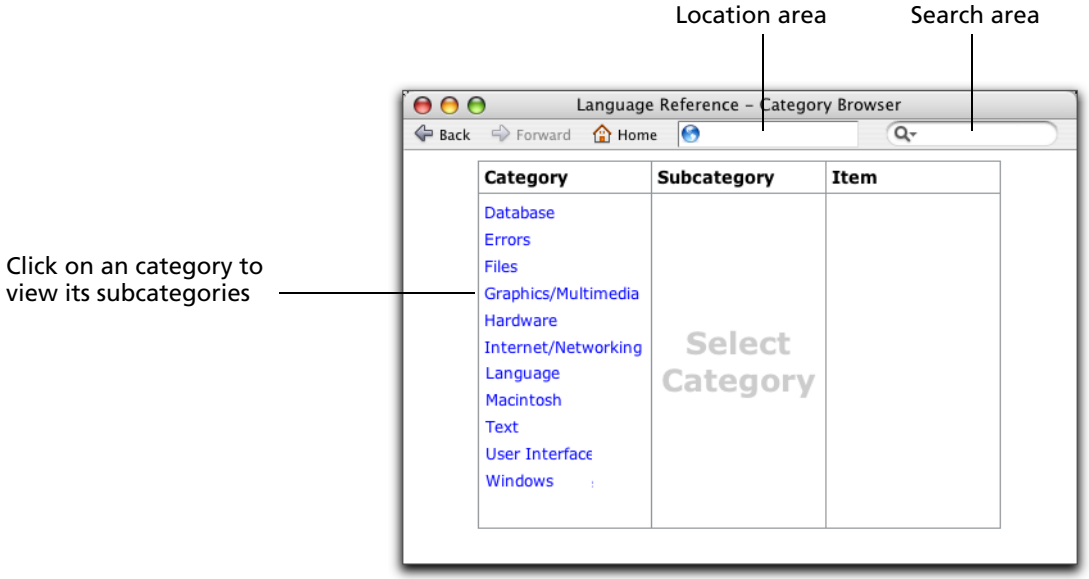
This icon means that the text to the right of it is important information that should not be overlooked.

This icon indicates that the text to the right pertains to Mac OS X only.

Using the On-Line Help

The REALbasic *Language Reference* is built-in to REALbasic. To access this language reference, choose Help ▶ Language Reference (F1 on Windows and Linux or ⌘-? on Macintosh) or press the Help key.

Figure 1. The On-Line Reference



Click a category name to view the subcategories and then click a subcategory to view its language items. Click on item to navigate to it.

Use the Arrows in the header area to move backward and forward through the items you have been browsing. The keyboard equivalents are Ctrl+[and Ctrl+] on Windows and Linux and Command-[and Command-] on Macintosh. The Home button takes you back to the page with the list of categories and subcategories.

Searching the Online Reference

Use the Location area in the header of the window to search the Online Reference. To search for a REALbasic language object (i.e., an item listed in the Browser), simply begin typing the name of the object.

As you type, REALbasic tries to guess the name of the object. If it has one guess, the guess appears in gray text. To accept the guess, press the Tab key. If it has several

guesses, three dots appear. Press Tab and a pop-up menu of choices appears. Use the up and down arrow keys to highlight the desired item and press Tab or Return to select it.

Use the Search area to search for any term, such as the name of a property, method or event. Press Enter to do the search. If it finds matching items, it will return a list of found items. For example, the following shows the results for a search on the term "multiline".

Figure 2. Search results in the Online Reference.

The Online Reference does not support multi-word searches, so you cannot use blank spaces in your search string.

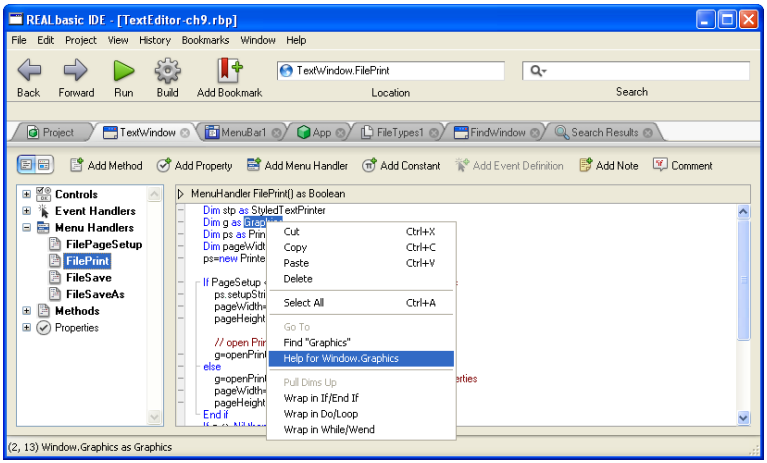
Context-Sensitive Help

The Online Reference is context-sensitive. If you select an interface control in a window then open the Online Reference, the Reference will open to information about the selected control.

The contextual menu in the Code Editor has a menu item for help. Highlight a command in your Code Editor for which you want help and then choose Help on... from the contextual menu. In the following example, the user has right+clicked on the term "Graphics" in the Code Editor[1]. The contextual menu appears and offers to look up this term in the Online Reference.

1. Macintosh users with one-button mice can Control-click

Figure 3. Looking up a term from the Code Editor.



Context-Sensitive Error Messages

When you attempt to compile REALbasic code, the application first checks your syntax. And, when the code executes, REALbasic may encounter a runtime error. If it finds an error, it displays a brief description of the error message in the Tips bar at the bottom of the REALbasic IDE window.

Using the HyperText Links in the Online Help

Any text that appears in the blue, underline style in the Online Reference is a *hypertext link*. Clicking on the text will switch the Online Reference to a page about the topic you clicked on.

Using the Code Examples

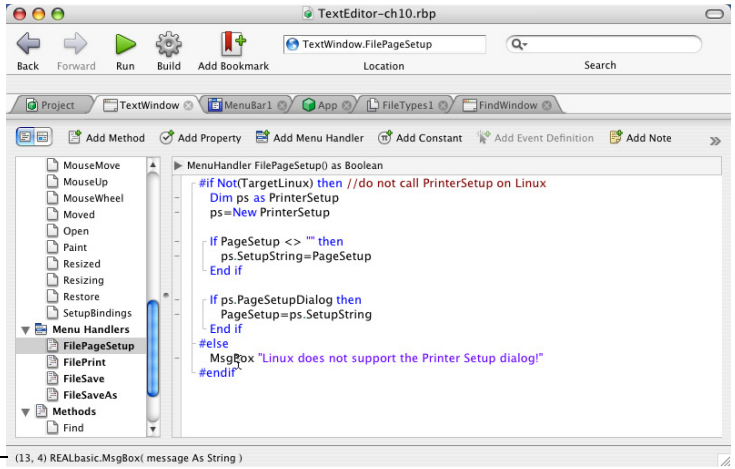
The Online Reference contains many code examples that you can use in your projects.

Some code examples omit irrelevant code for the sake of clarity. Omitted code is indicated by dots. Insert your own code in its place. Also, some examples include a Sub or Function statement, which is added automatically by the Code Editor. If copy and paste such an example into the Code Editor, you need to remove the duplicate statements.

Using Tips

As you work, the REALbasic IDE "watches" your activities and occasionally displays hints in the Tips bar. This is the bar at the bottom of the IDE window.

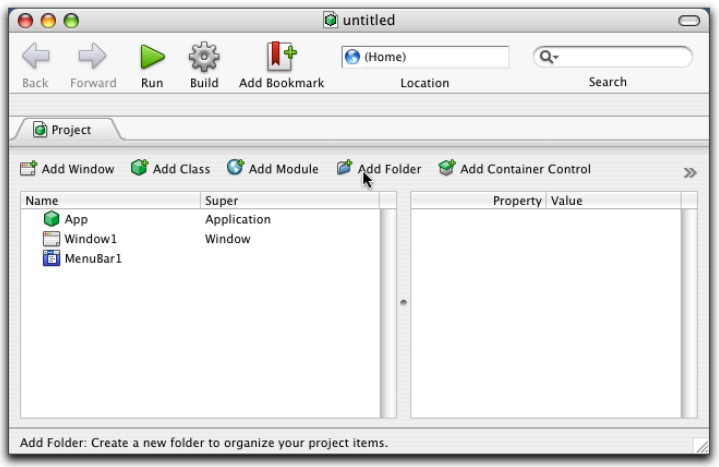
Figure 4. The Tips bar.



The Tips bar

For example, as you write code, tips remind you of the correct syntax for a command you are entering. This is shown in Figure 4. The insertion point is in the MsgBox command and the Tips bar shows its syntax. Also, the text in the Tips bar changes as you move the mouse around the IDE window. Simply place the mouse over an object to get information about it. For example, in Figure 5 the mouse pointer is over the Add Folder item in the toolbar and the Tips bar shows info on this item.

Figure 5. Help on the Add Folder item.



Electronic Documentation

All of the REALbasic documentation is available on the REALbasic CD and at our web site (http://www.realsoftware.com) and ftp site (ftp://ftp.realsoftware.com). These documents are available in PDF (Adobe Acrobat) form.

You can purchase printed copies of the *Tutorial* and *User's Guide* from REAL Software for an extra charge.

Our Support Web Page

Our support page is located at http://www.realsoftware.com/support. This page is the place to check for information on REALbasic. You'll find information on customer service, technical support, and links to mailing lists and news groups.

End User Web Sites

There are dozens of web sites created by other users dedicated to REALbasic. Check our web site for links to these sites.

REALbasic Developer

REALbasic Developer (http://www.rbdeveloper.com) is a magazine devoted to REALbasic programming tips and techniques. REALbasic Developer publishes articles by both experienced REALbasic users and the authors of REALbasic.

REALbasic third-party Books

The REALsoftware web site contains information on third-party books, web sites, magazines and other resources that offer support. Check for the most current information at http://www.realsoftware.com.

The REALbasic CD

The REALbasic CD contains the latest version of REALbasic, lots of examples, and documentation. We update the CD from time to time, adding updated documentation, examples, and other useful information. Most of the files on the CD are available at our web site. However, if you don't have an Internet connection or you just don't want to download hundreds of megabytes of files, you can always purchase an updated CD for a nominal fee. You can find the CD revision number of your CD in the Read Me file on the CD. You can check the purchase page of our web site to see if your CD is the latest version.

Our Internet Mailing Lists

We sponsor several Internet Mailing lists that give you the opportunity to ask questions, and share information with other REALbasic users via email. For more information on the available Internet Mailing Lists, see our support page at www.realsoftware.com/support.

Technical Support from REAL Software

REAL Software provides free email support for all users who have a current license. Older versions of REALbasic do not qualify for free email support. See the technical support page on our web site for information about our technical support policy.

We also have Developer Program that is available at an extra charge. It entitles you to priority technical support. See the support page at our web site (http://www.realsoftware.com/support) for more information or call us at 512-328-7325.

If you purchased REALbasic from an international distributor, you need to contact your distributor for technical support. You will find a list of all of our international distributors on our web site at http://www.realsoftware.com.

Contacting REAL Software

If you need to contact REAL Software, we can be reached in the following ways:

Phone	512 328-REAL (512 328-7325) from 9AM to 6PM Central Time, Monday through Friday
Fax	512 328-7372
email	Submit via REALbasic Feedback (Help ▶ REALbasic Feedback)
Mail	1705 South Capital of Texas Highway Suite 310 Austin, TX 78746

Reporting Bugs and Making Feature Requests

If you think you have found a bug in REALbasic or have a feature request, please let us know about it. The best way to report bugs or make feature requests is via Feedback page on the REAL Software web site. Feedback was designed to gather all the necessary information that helps us track down bugs and implement feature requests. For each bug or feature request reported, you will receive a confirmation message via email with a tracking number you can use to check on the status of your bug report or feature request. Once we close the issue, we will email you with the reason the issue was closed (e.g., the bug has been fixed for the next release, the feature will be implemented in the next release, it's not a bug after all, etc.).

You can access Feedback directly from within REALbasic. Just choose Help ▶ REALbasic Feedback and your default web browser will open the Feedback page.

If you don't have an email account, you can send us your bug reports and feature requests via regular mail to our mailing address or fax them to us.

CHAPTER 2

Getting Started with REALbasic

Building a simple application with REALbasic can take just a few minutes. First, you create your user interface, which consists of menus and windows filled with interface controls. Once you have created the interface, you use REALbasic's programming language to make the interface do what you want it to do when you want it to do it!

This chapter will give you an overview of the important concepts you need to understand the REALbasic development environment and how to work with projects.

Contents

Concepts

There are a few important concepts you will need to understand in order to develop applications with REALbasic. You should also be very comfortable with the graphical user interface your computer uses. If you are not, it would be a good idea to spend some time getting familiar with it before you begin using REALbasic. Otherwise, you may find many of the references in this documentation confusing.

Applications are Driven by Events

Before computers used graphical user interfaces, applications ran by simply executing a series of programming code statements starting with the first statement and ending with the last. Interfaces were all character-based. A menu was just a numbered list of commands that the user selects from to instruct the application to do a task. Most of the time, the application was just sitting there waiting for the user to make up his mind. When the user finally chose a command (perhaps by selecting the number next to the menu item and pressing the Enter key) the application would take whatever action was associated with the chosen command. When the user pressed the Enter key, an *event* occurred. In other words, something happened to which the application can respond.

Now that desktop computers use a graphical user interface, users have a far more intuitive way to interact with applications. However, one thing hasn't changed: applications are still driven by events. The difference is that back in the old days there were very few events the application had to worry about responding to. The old-fashioned application was always in a modal state: It only had to respond to the limited number of choices it presented to the user. With a graphical user interface, many more choices and ways of interacting with the computer are available. The user might choose a menu item, click on a button, or type in a field. Also, the applications themselves may cause events to occur that were not directly caused by the user. For example, when a window opens, an event occurs (the window opened). When a window is moved or resized, an event occurs.

Fortunately, REALbasic makes it easy to deal with all of these different events. You can easily find out which events each part of your application's interface can respond to. Making your application respond to an event is as easy as locating the object that will receive the event, selecting the event, and entering the instructions (using REALbasic's programming language) you want the object to follow when the event occurs. Later on, you will learn about events in more detail. For now, it's just important to understand the concept of event-driven programming.

Developing Software with REALbasic

If you have written computer programs using traditional programming languages, you already know that the process of development is three steps: write some code, compile the code (turning the code into something the computer can really understand), and test your application. When you find a problem in your application, you start the process over again. Developing software applications with REALbasic isn't much different than that. The big difference is how often you go through this process. Compilers for traditional languages can take several minutes or more to com-

pile an application before you can begin testing. Consequently, you spend a lot of time writing code before compiling to avoid waiting for the compiler. REALbasic's compiler is so fast that you will find you can make a small change to your code and immediately run it to make sure the change you made works as expected.

Like traditional programming language compilers, REALbasic's compiler will stop if it finds a syntactical error in your code and inform you what the error is so you can fix it. But unlike traditional compilers that require you to track down the line of code where the error occurred, REALbasic's compiler takes you right to the point in your source code where the error occurred. It then displays the error message just below the line of code that caused the error. It puts you right where you need to be to fix the problem.

If you have used traditional programming languages, you will find developing applications with REALbasic to be easier, faster and more fun.

The Development Environment

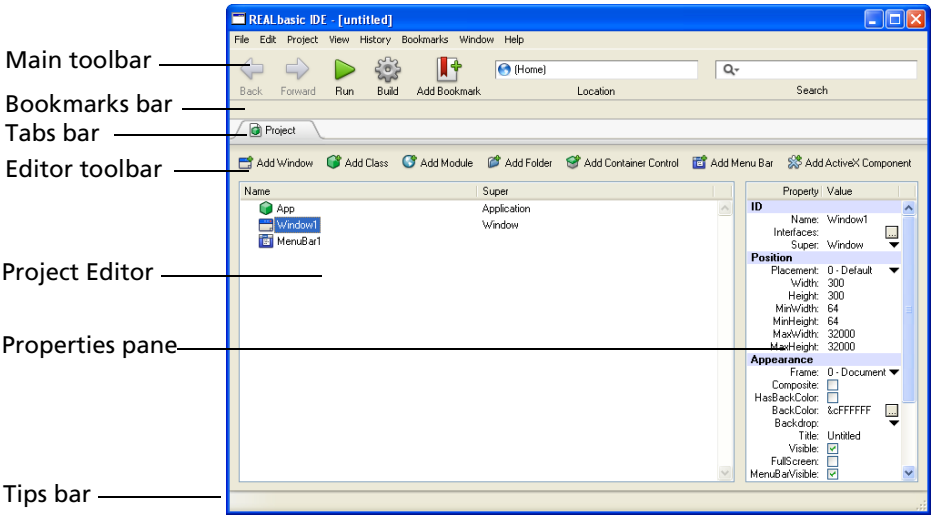
REALbasic is an *Integrated Development Environment* (IDE) which means that it contains everything you need to build an application. An interface builder, code editor, compiler, and debugger are all integrated into one package. In traditional programming languages, these items would each be a separate application.

The REALbasic IDE Window

The REALbasic IDE window organizes all the components of your application into a series of screens. The interface should be very familiar to anyone who is used to working with an internet browser that has the ability to use multiple tabs. This includes Mozilla or Firefox on Windows, Linux, and Macintosh and Safari on Macintosh.

When you start REALbasic, the main IDE window appears:

Figure 6. The REALbasic IDE window.



The IDE window resembles an internet browser window. The Main Toolbar contains controls for navigating among parts of your project. The Back and Forward buttons allow you to redisplay previously viewed screens, the Location field lets you go to an item by name, and the Search area lets you search for specific objects throughout your project.

With the Add Bookmark button, you can add frequently used items to the Bookmarks menu or the Bookmarks bar. You can navigate to bookmarked items simply by choosing its name from the Bookmarks menu or clicking on it in the Bookmarks bar.

The History menu tracks the series of screens that you have worked on. You can redisplay a screen by choosing its name from the History menu.

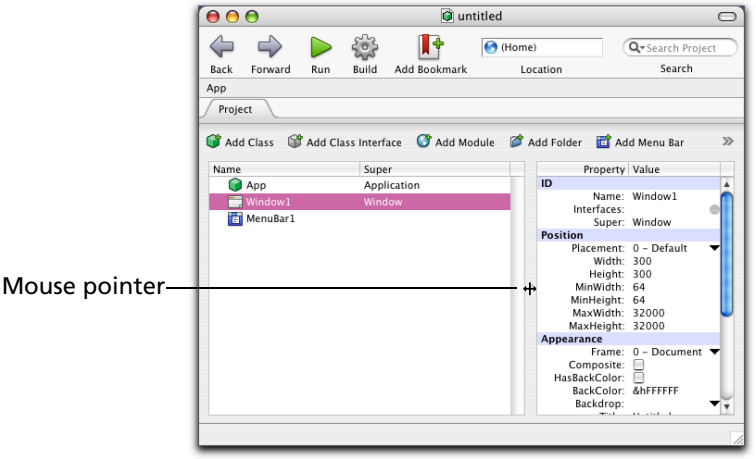
The body of the window displays parts of the project organized into screens. Each screen is labeled with a tab in the Tabs bar. Except for the Project Editor, each tab has a close box that you can use to close that screen. Closing a close box only closes the screen; it does not delete the item.

When you first launch REALbasic, only one screen is available, the one for the Project Editor. Just below the Tabs bar is a toolbar that belongs to the editor that is displayed. You use it to add or manipulate items in the editor you are viewing. The Editor toolbar changes depending on which editor you are on.

Each screen is divided into two or more *panes*. In the case of the Project screen, it is divided into the Project Editor and the Properties pane. Panes are separated by dividers that can be moved to the left or right by holding down the mouse button on the divider and dragging to the left or right. To resize the panes, move the pointer to the divider until the mouse pointer changes to an Arrow pointer that

points to the left and right. Then hold down the mouse button and drag in either direction.

Figure 7. The Properties pane being resized.



You can maximize the usable area in an editor by selecting the View ▶ Editor Only menu command. This menu command hides the Main toolbar, the Bookmarks bar, and the Editor toolbar, leaving only the Tab bar above the editor. In some cases, it also minimizes the space taken up by secondary panes in the editor area. When the Editor Only menu command is selected, REALbasic places a checkmark to its left. Selecting the Editor Only command again changes the IDE to its previous configuration.

The following sections give an overview of each type of screen.

The Project Editor

A *project* is the collection of items that make up the application you are developing. The Project Editor organizes all the major components of the application. You add these items to the application from the Project and you can access each item from this screen.

For example, each of the windows that makes up your application will be listed in the Project Editor. Some of the other items that might be listed in the Project Editor are classes, modules, menubars, pictures, sounds, databases, and movies.

You use the Project Editor toolbar or the Project ▶ Add submenu to add items to the project. You will learn more about projects in the next chapter.

The Project Editor displays a list of these elements to give you easy access to them. You click on a project item to go to an editor for that item or, if there is no editor, a viewer.

The Properties Pane

When you select an item in the Project Editor by clicking on it, the Properties pane (the pane to the right of the divider) shows properties that belong to the selected

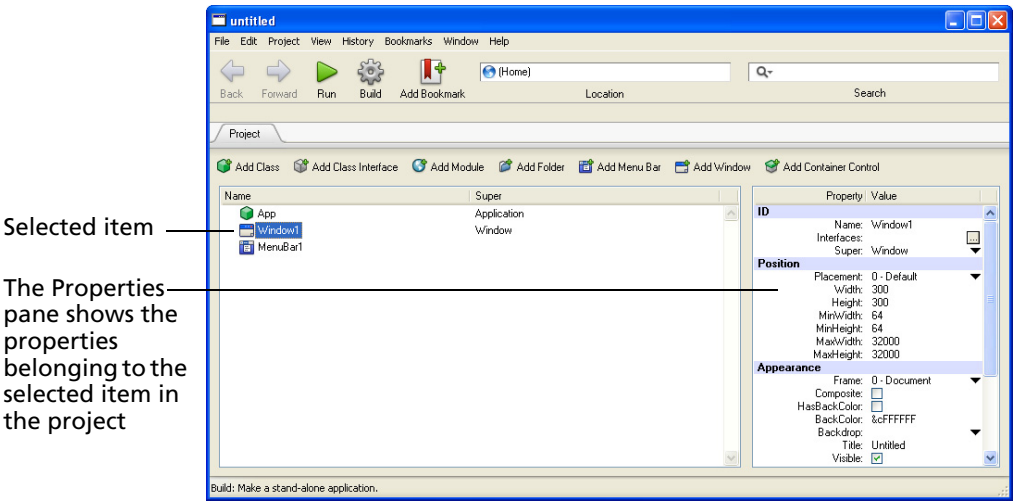
item. For example, in Figure 7 the Window1 item is selected in the Project Editor, so its properties are shown in the Properties pane.

Properties are values that are owned by an item. They characterize the item. Some examples of items that have properties are a window, a menu item, and a control in a window. For example, a window has a Title property that holds the text that is shown in the window's Title bar. A window also has Width and Height properties that store the window's size. A window's Left and Top properties describe the position of its top-left corner.

The Properties pane displays all of the properties that *can be modified in the IDE* for the currently selected item. This is an important point because some objects have properties that can be modified only by your programming code. An item may also have properties that cannot be modified or can be modified only from the IDE. The appearance of the Properties pane depends on which object is selected.

For example in Figure 8, Window1 is selected (highlighted) in the Project Editor and the Properties pane shows various properties of this window, such as its name, frame type, height, and width. For more information about the Properties pane, see the section"Using a Window's Properties Pane" on page 71 and "Changing a Control's Properties with the Properties Pane" on page 95.

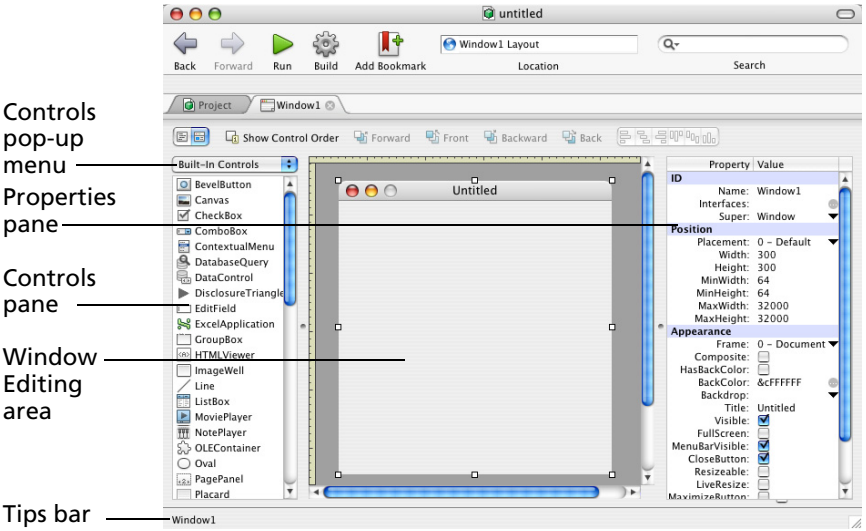
Figure 8. The Project Editor with Window1 selected.



The Window Editor

You use the Window Editor to design the user interface for each window in your project. To access the Window Editor for a window, double-click the window's name in the Project Editor. REALbasic adds a new tab to the Tab bar and displays the editor for that window in that screen. The Window Editor for the default window, Window1, is shown in Figure 9.

Figure 9. An example window displayed in its Window Editor.



Notice that the Tabs bar now has two tabs. You can return to the Project Editor by clicking on its tab. Also, the Location field has changed to show the name of the editor. For a Window Editor, the name of the screen consists of the name of the window followed by "Layout." You can always navigate to this screen by entering its name in the Location area. This Window Editor can be closed by clicking its tab's close box, but the Project Editor cannot be closed.

The window being edited is in the center of the screen and its properties are shown in the Properties pane on the right. These are the same properties as you saw from the Project Editor when you clicked on Window1 item in the Project Editor.

The dividers on either side of the editing area can be dragged to the left or right to resize the editing area within the IDE window. When the mouse pointer is over a resizing hot point in a divider (indicated by a dot in Figure 9), it changes to a resizing pointer with arrows pointing to the left and right. Drag to resize the editing area. Resizing is shown in Figure 7 on page 31.

The window editing area has horizontal and vertical scroll bars that enable you to view different areas of the window without resizing the IDE window or resizing the editing area. You can also reposition the window in the editing area by dragging its title bar.

The Controls Pane

The list on the left side of the Window Editor screen shows the names of REALbasic's built-in controls. *Controls* are interface elements such as buttons, CheckBoxes, text entry fields, lists, tab panels, pop-up menus, and movie players.

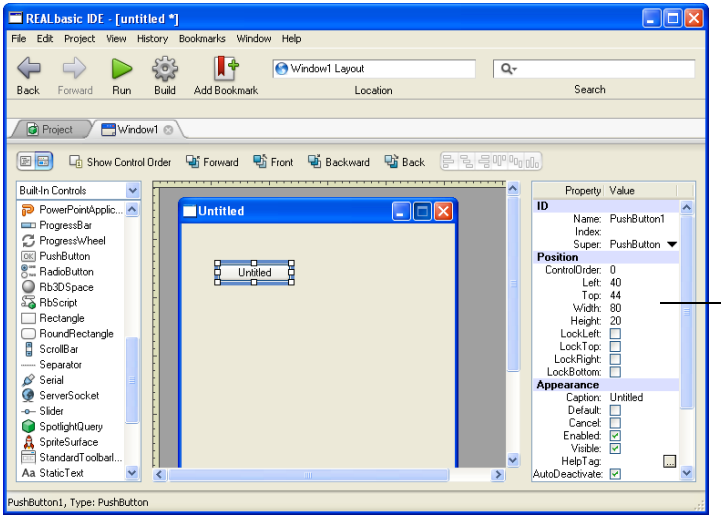
You use the Controls pane to add controls to the window that you are designing. To add a control to a window, you can double-click on the control, drag it from the Controls pane to the window editing area, select the control and then press the

Enter key (Return on Macintosh), or select the control and then drag an area in the Window Editing area in the location, size, and shape that you want.

When you add a control to a window, an instance of that type of control appears in the Window editing area and the Properties pane changes to show the properties for that control.

For example, in Figure 10, a PushButton control has been added to the window. Its resizing handles indicate that it is selected.

Figure 10. A PushButton control added to Window1.

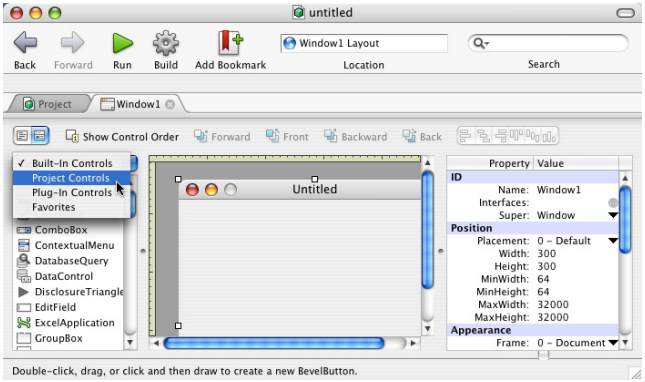


Since the PushButton control is selected in the Window Editor, the Properties pane has changed to show the properties of the PushButton. To redisplay the window's properties, deselect the PushButton by clicking on the surface of the window.

The Controls Pop-up Menu

The pop-up menu above the list of controls enables you to populate the Controls list with one of four types of controls:

Figure 11. The Controls pop-up menu.

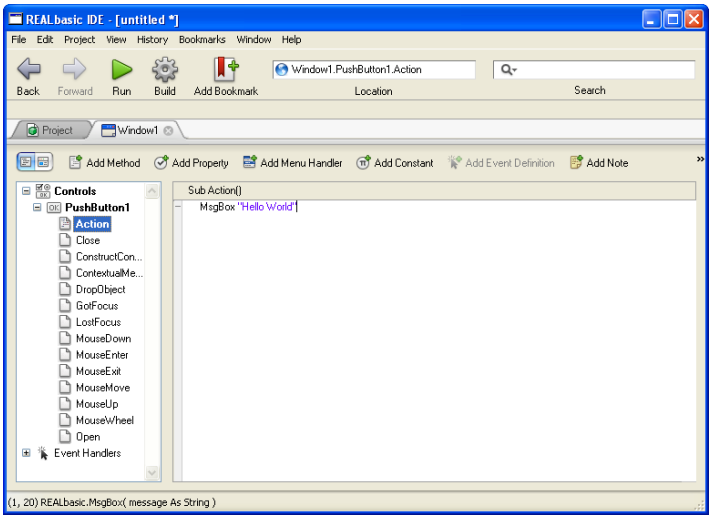


- **Built-in Controls:** The controls that are built into REALbasic. This is the default choice. The built-in controls are shown in Figure 10. For more information on REALbasic's built-in controls, see the section "Interacting with the User Through Controls" on page 84.
- **Project Controls:** Custom controls that are based on built-in controls. Project controls are also listed as items in the Project Editor. For information on creating custom controls, see the section "Understanding Subclasses" on page 421 and the procedures for creating subclasses based on controls in Chapter 9 on page 419.
- **Plug-in Controls:** Controls that you added to REALbasic by installing plug-ins. Third-parties can market custom controls in the form of plug-ins that are installed by placing the plug-in in the Plugins folder in the REALbasic folder. This list is empty if you have no third-party plug-in controls installed.
- **Favorite Controls:** Controls from any of the three other types of control lists that you have marked as Favorites. For more information, see the section "Favorite Controls" on page 85.

The Code Editor

Use the Code Editor to add programming code to items in your project, such as controls, menu items, and windows. The Code Editor has a browser area that makes it easy to locate the object and view all the events the object can receive. The Code Editor is shown in Figure 12.

Figure 12. The Code Editor for a window.

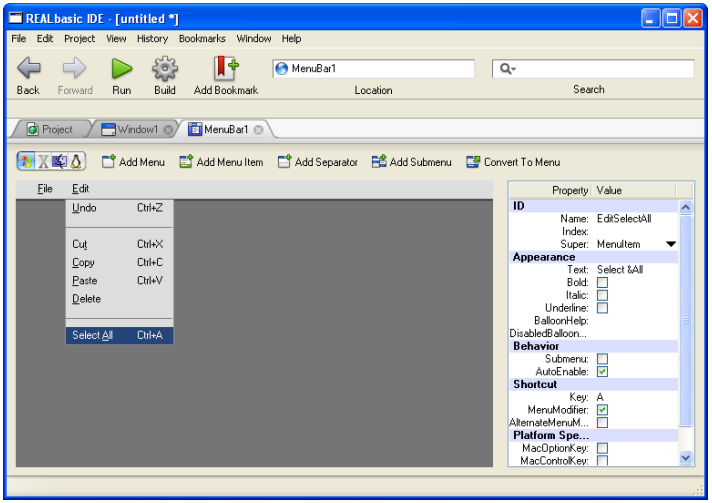


For a detailed description of the Code Editor, see the section "Using The Code Editor" on page 219.

The Menu Editor

You use the Menu Editor to create the menus and menu items that will be displayed when your application runs. The Menu Editor is shown in Figure 13.

Figure 13. The Menu Editor.



When you click on a menu or menu item in the Menu Editor, the Properties pane changes to show the properties of the item. The Text property of the menu item is the text that is displayed by the menu item. You can also assign keyboard shortcuts to menu items and even create sub-menus (a menu item that is actually just another

menu). REALbasic adds File, and Edit menus for you by default. For Macintosh, it also adds the Apple and Application menus.

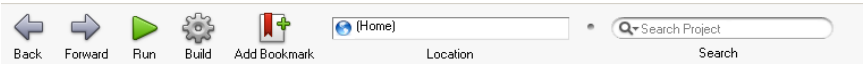
The row of icons on the left side of the Menu Editor toolbar allows you to preview the look of the menu system under every possible operating system on which you can build the application. Click on one of the View Mode buttons to preview the menu system on that platform.

Figure 14. The Menu Editor's View Mode buttons.

The Main Toolbar

The REALbasic Main Toolbar has items for navigating among editors in your project, for testing your project, and for building a standalone application. Figure 15 shows the default Main Toolbar.

Figure 15. The default IDE Main Toolbar.



- **Back and Forward buttons:** The Back and Forward buttons work exactly like the Back and Forward buttons in a Web browser; they move backwards and forwards among IDE screens in the order that you viewed them. This list of screens is also displayed as items in the History menu. You can use the History menu to jump to any previously visited page.
- **Run button:** Click the Run button to compile the project and test it. When you click Run, REALbasic attempts to compile your application. If the project compiles successfully, the application launches in its own window and REALbasic adds a new screen in the IDE window for real-time debugging that's called the "Run" screen. Click Stop in the Run screen's Editor toolbar to halt execution of the application.

  You can switch to another editor in the IDE and continue working on your application without quitting the debugging application, but your changes won't be reflected until you stop the debugging application and rerun it.

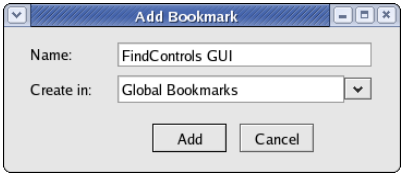
  If REALbasic finds any syntax errors, the attempt to compile and launch the application will stop and REALbasic will display the errors in an Errors screen or in the Code Editor in which it found the error. You need to fix all the syntax errors to allow REALbasic to successfully compile and run the application. For information about fixing syntax errors, see the section "Syntactical Bugs" on page 500.

  For information on testing the application within the IDE, see Chapter 11, "Debugging Your Code" on page 499.
- **Build Button:** Click the Build button to build a standalone (double-clickable) application based on the current Build settings. The difference between "Run" and

"Build" is that the result of a "Build" is a standalone application that doesn't need the REALbasic application at all. A "Run" is a test-run of your program inside REALbasic. See Chapter 14, "Building Stand-Alone Applications" on page 547 for information on Build Settings and building standalone applications. The attempt to build will fail if there are any syntax errors.

- **Add Bookmark button:** Click the Add Bookmark button to add the current item to either the list of Bookmarks in the Bookmarks menu or to the Bookmarks bar. An item can be an entire editor, a method, a property, a control, a file type set, and so forth. This list is displayed in the Bookmarks menu and the Bookmarks bar is just above the Tabs bar and below the Main Toolbar. When you choose this menu item, the dialog box shown in Figure 16 appears:

Figure 16. The Bookmarks dialog box.



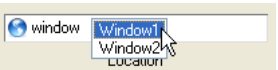
Choose Global Bookmarks from the Create In drop-down list to add the item's name to the Bookmarks menu. It's called "global" because the bookmark will appear in all of your projects. Choose Local Bookmarks bar to place it in the current project's Bookmarks bar. Each project has its own Bookmarks bar.

If you have created bookmark folders, they will also be offered as choices in the Create In drop-down list. This enables you to place a new bookmark directly in a folder.

You can navigate directly to a bookmarked item by choosing its name from the Bookmarks menu or clicking on its name in the Bookmarks bar.

- **Location area:** Use the Location area in the same way you use the URL area in a Web browser. Enter the name of an item and press Enter (Return on Macintosh) to go to it. The Location area uses the autocomplete feature of REALbasic. As you type, REALbasic attempts to complete the term that you are typing. Its guess is shown in gray. If it has more than one guess, it shows three dots and you can press Tab to see a contextual menu of the possibilities.

Figure 17. Using Autocomplete in the Location area.



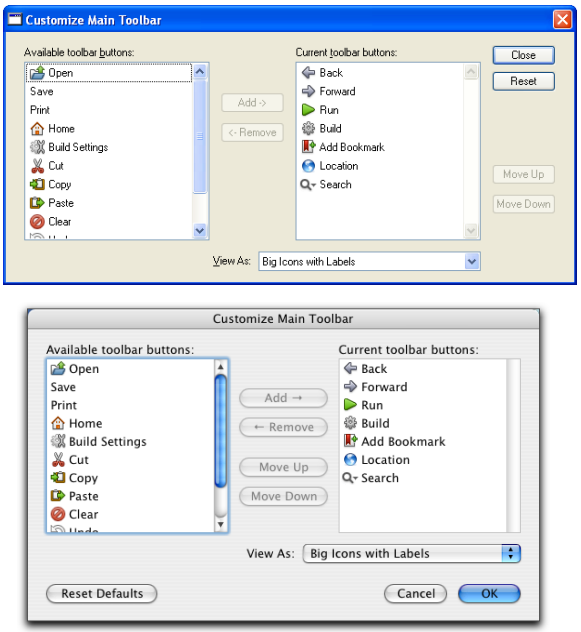
For more information about autocomplete, see the section "Autocomplete" on page 232.

- **Search area:** Use the Search area to find items in your code. A pop-up menu lets you choose among searching the whole project, the current item, or the current method (if applicable).

Customizing the Main Toolbar

If you like, you can add, remove, or reposition the items in the Main Toolbar. To do so, choose View ▶ Main Toolbar ▶ Customize. The Customize Main Toolbar dialog box appears.

Figure 18. The Customize Main Toolbar dialog box.



The Customize Main Toolbar dialog box uses a "mover" interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. Listed on the left are optional items that can be added to the toolbar.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (shown in Figure 18).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down or click the item you want to move, drag it to the desired location, and then drop it between two items. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down list. Your choices are:

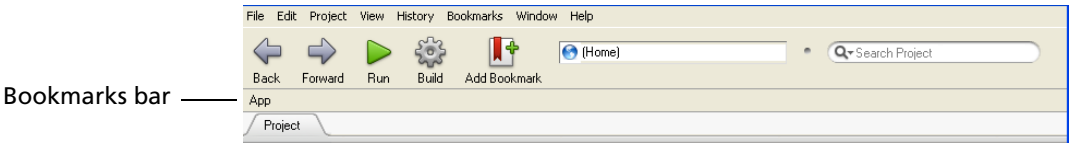
- Big icons with labels.
- Small icons with labels,
- Big icons (no labels),
- Small icons (no labels),
- Labels only.

- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or Factory Defaults button (Macintosh).

The Bookmarks Bar

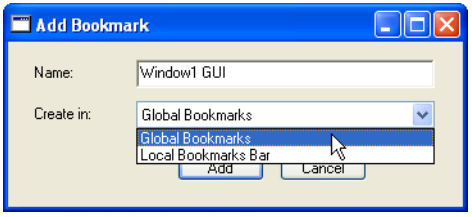
The Bookmarks bar, which is just below the Main toolbar, displays the locations that you have added to the local bookmarked locations. You can go to a bookmarked location simply by clicking on the item in this toolbar.

Figure 19. The Bookmarks bar.



To add an item to the Bookmarks bar, click the Add Bookmark button in the Main Toolbar or choose Bookmarks ▶ Add Bookmark. The Add Bookmark dialog box shown in Figure 20 appears.

Figure 20. The Add Bookmark dialog box.



By default, the Create In drop-down list offers two choices:

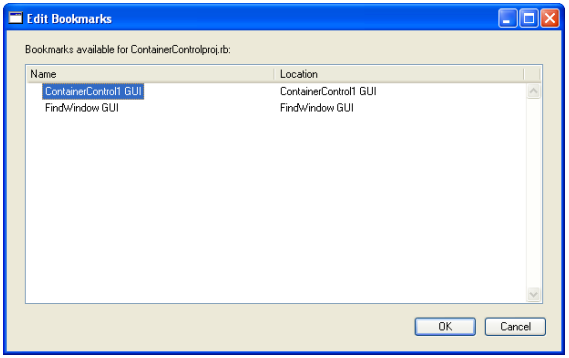
- **Global Bookmarks:** Global bookmarks appear as items in the Bookmarks menu. They are global in the sense that they appear in all of your projects.
- **Local Bookmarks bar:** Local bookmarks appear in the Bookmarks bar in the project's IDE window. When you add a bookmark to the Bookmarks bar, it is local to that project; it does not appear in the Bookmarks bar of your other projects.

  You can also add methods and properties to the Bookmarks bar by dragging the name of the method or property from the Code Editor browser list to the Bookmarks bar. When the item to be added to the Bookmarks bar is over the Bookmarks bar, it shows a selection rectangle, indicating that it can accept the dragged item.

If you have created bookmark folders for the Bookmarks menu, they will also be offered as choices in the Create In drop-down list. This enables you to place a new bookmark directly into a folder. When you add a item to a folder, the Bookmarks menu shows the folder's name as a menu item and the items in the folder as submenu items.

You can also modify existing bookmarks. To do so, right+click (Control-click on Macintosh) on the Bookmarks bar to display the Bookmarks bar's contextual menu and choose Customize. The Edit Bookmarks dialog box appears:

Figure 21. The Edit Bookmarks dialog box.



With the Edit Bookmarks dialog box, you can modify the names and locations of your bookmarks. You can edit the name to make it easier to recognize.

Locations use the "dot" syntax to specify the project and Editor, and item names. That is, the syntax is *projectname.editorname.itemName*. If it is a control on a window, the dot syntax is extended: *projectname.editorname.controlname*. For windows, the window name refers to the Code Editor view and "*window name* Layout" refers to the Window Editor for the window. When you create the bookmark, REALbasic suggests the name that will work, so you can simply accept the default name.

To modify either the name or location, click twice in the text to get an insertion point and type the replacement text.

REALbasic IDE Menus

The REALbasic IDE has the following menus: File, Edit, Project, View, History, Bookmarks, Window, and Help. The Macintosh version adds the standard Apple and REALbasic menus.

The File Menu

The File Menu has menu items for creating, opening, and saving items.

- **New Project:** Creates a new project. It first gives you a choice of a new Desktop Application, Console Application, or an application based on a user-defined template. You can also specify a custom template (or starting point) for the new project. When you make your choice, REALbasic opens a blank IDE window for the

project. You can have more than one project open at the same time. For more information about Desktop and Console applications and custom templates, see the section "Creating A New Project" on page 52.

- **New Window:** Opens a new REALbasic IDE window for the current project. It is a duplicate of the current IDE window, but you can, of course, navigate to different parts of the project in the new window. Use this menu command to view two or more editors in your project simultaneously. Changes in one window cause the others to be updated.
- **IDE Scripts:** Has a submenu with the New IDE Script command and the names of any existing IDE scripts. New IDE Script opens an IDE Script Editor window that enables you to write code that automates the REALbasic IDE itself. You can either type in the code or have the IDE Script Editor record your actions. In the latter case, click the Record button, perform the actions that you want the script to automate, and then click it again to stop the recording process. It will generate the code and add it to the Script Editor. In either case, you use the RBScript language. See the entry for RBScript in the *Language Reference* for the list of IDE Script commands.
- **Open:** Displays an open-file dialog box that allows you to open an existing REALbasic project. You can open either standard REALbasic projects or projects saved in XML format. The Save and Save As menu commands give you a choice of formats in which to save your project. When you select a project, it opens in its own IDE window, keeping the current IDE window open. This menu command enables you to have several projects open at once.
- **Open Recent:** Open Recent has a submenu of recently opened REALbasic projects. When you choose a project from the submenu, it opens in its own REALbasic IDE window keeping the current IDE window open.
- **Close Tab:** Closes the tab panel that is currently in front. You can also close a Tab by clicking the Tab's close box. If you close the Project tab, REALbasic will try to close the project itself. If you have unsaved changes, REALbasic will give you a chance to save your changes.
- **Close Window:** Closes the current IDE window. If you have unsaved changes to the project, REALbasic will give you a chance to save your changes.
- **Save:** Saves the current project or the project whose IDE window is active. A drop-down list enables you to save in the standard project format or in XML. The standard REALbasic project format is the default and is the recommended format if you have no special need to work with the XML. If no changes have been made since the last save operation, this menu item is dimmed.
- **Save As:** Saves the current project or project whose IDE window is active under a new name. A drop-down list enables you to save in the standard project format or in XML. The standard REALbasic project format is the default and is the recommended format if you have no special need to work with the XML.

- **Revert to Saved:** Reverts the current project or the project whose IDE window is active (i.e., in the front) to its last saved state.
- **Import:** Opens an open-file dialog box that enables you to import an item into the project. You can import any type of item that can appear in the Project Editor. This includes object such as windows, classes, modules, pictures, sounds, and movies. You can also import items by dragging them from the desktop to the Project Editor.
- **Export *Item*...**: Exports the contents of the currently selected item. The Save-file dialog box gives you a choice of the REALbasic format for the item or the XML format.
- **Export Localizable Values...**: Exports all of the dynamic string constants in the application to a file. This file is readable by REAL Software's free localization application, Lingua. Lingua is the utility that you use to localize REALbasic applications. When you are finished localizing the string constants in Lingua, you import this file back into your application with either the File ▶ Import command or by dragging it into the Project Window. For information about Lingua, see the section "Using Lingua to Localize your Application" on page 311.
- **Page Setup:** Displays the Page Setup or Print Setup dialog box for your operating system.
- **Print**: Prints the project's properties, which are shown in the Properties pane for the App class, and all the project's code. For information on the project's properties, see the section "Customizing the Standalone Application's Properties" on page 550.
- **Print *Item*:** Prints the selected item according to the current Page Setup settings.
- **Exit or Quit:** Closes all open IDE windows and Quits the REALbasic IDE application.

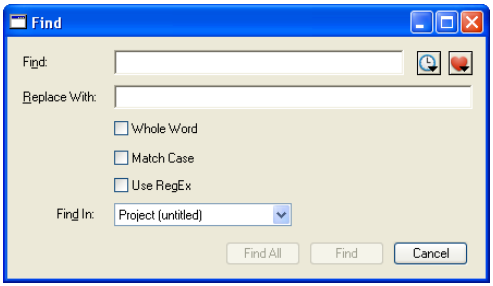
The Edit Menu

The Edit menu has the standard editing commands and some REALbasic-specific commands for working with the Code and Window editors.

- **Undo:** Undoes the last action. If this is impossible, this item changes to Can't Undo.
- **Redo:** If you have just chosen Undo, the Redo menu item becomes active, offering to redo the action that was undone. In other cases, this menu item is dimmed.
- **Cut:** Cuts the selected text or item and places it on the Clipboard.
- **Copy:** Copies the selected text or item and places it on the Clipboard.
- **Paste:** Pastes the item on the Clipboard at the insertion point (text) or into the current object, such as a window in a Window editor.
- **Delete:** Deletes the selected text or item without putting it on the Clipboard.

- **Select All:** Selects all of the text or all the items in the current editor. It does not place the items on the Clipboard.
- **Deselect All:** Deselects all the currently selected items.
- **Comment:** Changes the current line or line in which the text insertion point is located into a nonexecutable comment line. If the current line is already a comment, this menu item changes to Uncomment. The Uncomment menu item changes the line back to a line of executable code. These two menu items are equivalent to the Comment and Uncomment buttons in the Code Editor toolbar.
- **Encrypt *Item*:** Displays a dialog box for encrypting the selected project item. An encrypted item cannot be viewed or edited in its editor. If the selected project item is already encrypted, this menu item changes to Decrypt *Item*. Decrypt *Item* presents a dialog box enabling you to decrypt the item. The Encrypt and Decrypt Item menu items duplicate the functionality of the optional Encrypt and Decrypt buttons that can be installed in the Project Editor toolbar.
- **Duplicate:** Duplicates (copies and pastes) the selected text or item.
- **Arrange:** Displays a submenu for changing the control order of objects in a window, Bring Forward, Bring to Front, Send Backward, Send to Back. These choices affect the order in which the user can move from one control to another by pressing the Tab key. It also affects the display order for controls that are overlapped. Available only for Window Editors. These submenu items duplicate the functionality of the Window Editor toolbar.
- **Align:** Displays a two-part submenu for aligning objects in a window, The menu items above the separator align two or more objects by their left edges, right edges, top edges, and bottom edges. The items below the separator are for spacing three or more objects evenly, in either the horizontal and vertical directions. These items are available only for Window and Container Control Editors. These submenu items duplicate the functionality of the Window Editor toolbar.
- **Find:** Displays the Find and Replace dialog box for finding project items.

Figure 22. The Find dialog box.



For information on the Find dialog, see the section "Find and Replace dialog" on page 239. You can also find project items using the Search area in the Main toolbar.

- **Options:** Displays the Options dialog box for setting application-wide preferences. On Mac OS X, this item appears in the REALbasic menu as "Preferences".

The Project Menu

The Project menu has items for adding items to the project, testing it within the IDE, debugging code with the debugger, and building a standalone application.

- **Add:** The Add menu item has a submenu for adding a window, a class, a class interface, a container control, a module, a folder, a menu bar, a file type set, an ActiveX component (Windows OS only), and a database. The Project Editor's toolbar duplicates much of the functionality of the Add menu. Below a separator, the Add menu lists items that are particular to the type of editor that is currently shown. These menu items duplicate the functionality of the editor's toolbar.
  - **Window:** When Window's Code Editor is shown, the Add menu item has submenu items for adding methods, properties, computed properties, constants, menu handlers, and notes. These items duplicate the functionality of the Code Editor toolbar. When a Window Editor's Layout Editor is shown, these menu items are not available; the commands that correspond to a Window Editor's toolbar are in the Edit menu's Align and Arrange submenu items.
  - **Menubar:** When a Menu Editor is shown, the Add menu item has submenu items for adding a menu, menu item, submenu, and a separator. These submenu items duplicate the functionality of the Menu Editor toolbar.
  - **Module:** When the Code Editor for a module is shown, the Add menu item has a submenu for adding methods, properties, event definitions, constants, and notes. These submenu items duplicate the functionality of the Code Editor toolbar for a module.
  - **Class Interface:** When the Code Editor for a class interface is shown, the Add menu item has a submenu for adding methods and notes. These submenu items duplicate the functionality of the Code Editor toolbar for a class interface. Other items that are normally available for a Code Editor are not appropriate for a class interface. For more information about class interfaces, see the section "Class Interfaces" on page 458.
  - **Container Control:** When the Code Editor for a Container Control editor is shown, the Add menu item has submenu items for adding methods, properties, event definitions, constants, menu handlers, and notes. These items duplicate the functionality of the Code Editor toolbar. When the Container Control's Layout editor is shown, the Arrange and Align submenus of the Edit menu contain the items that duplicate the functionality of its editor toolbar. For information about container controls, see the section "The Container Control" on page 139.
- **Turn Breakpoint On:** The Turn Breakpoint On menu item sets a breakpoint in the Code Editor in the selected line or the line in which the text insertion point is located. This is equivalent to clicking in the left margin in the Code Editor to set the breakpoint. Lines on which you can set a breakpoint are indicated by a dash in the first column of the Code Editor. When a Breakpoint is set in a line, this menu

item changes to Turn Breakpoint Off. For information on setting breakpoints, see the section "You Have Set A Breakpoint In Your Code" on page 506.

- **Clear All Breakpoints:** The Clear All Breakpoints item removes all breakpoints in the project. It is available only for Code Editors and only when at least one breakpoint is set.
- **Break on Exceptions:** If the Break on Exceptions menu item is selected (has a checkmark to its left), the REALbasic compiler will break into the Debugger when it encounters a runtime exception error. For information on the Break on Exceptions option, see the section "Runtime Exception Errors" on page 510.
- **Run:** Attempts to compile the application and launch it. If REALbasic is successful, the application is launched and a new Debugger screen labeled "Run" is added to the IDE window. If the attempt to compile is unsuccessful, REALbasic will display the syntax errors that prevented compilation. This item is equivalent to the Run button in the Toolbar. Use Run to test and debug your application. For more information, see Chapter 11, "Debugging Your Code" on page 499.
- **Run Remotely:** The Run Remotely item has submenus that enable you to test the application on another computer. Most often you will use the Run Remotely feature to test the application on another platform. It has at least one submenu item, Setup, and items for each remote computer that is set up for remote debugging. The Setup menu item displays a dialog box in which you can choose the remote machines on which you will debug. Each target machine must have the Remote Debugger Stub utility configured and running. Remote Debugging is available only in the Professional version of REALbasic. For more information, see the section "Remote Debugging" on page 514.
- **Pause:** This menu item pauses execution of the test application in the debugger. It is enabled only if you are testing the application in the debugger. For more information on this command, see the section "Controlling Execution" on page 505.
- **Resume:** Available if you have paused execution. Continues execution from the breakpoint line without further interruption. This menu item is enabled only if you have paused at a breakpoint.
- **Step:** Use the Step command with the Debugger to step through your code line by line. Step has submenu items for stepping over, into, and out of a method. This duplicates the functionality available in the Debugger toolbar. For information on the Step options, see the section "Controlling Execution" on page 505.
- **Build Settings:** Displays the Build Settings dialog box. Use this dialog box to choose the platform or platforms for which you will build standalone applications. You can build for all platforms that REALbasic supports simultaneously. For information on the Build Settings dialog box, see the section "Build Settings" on page 550.

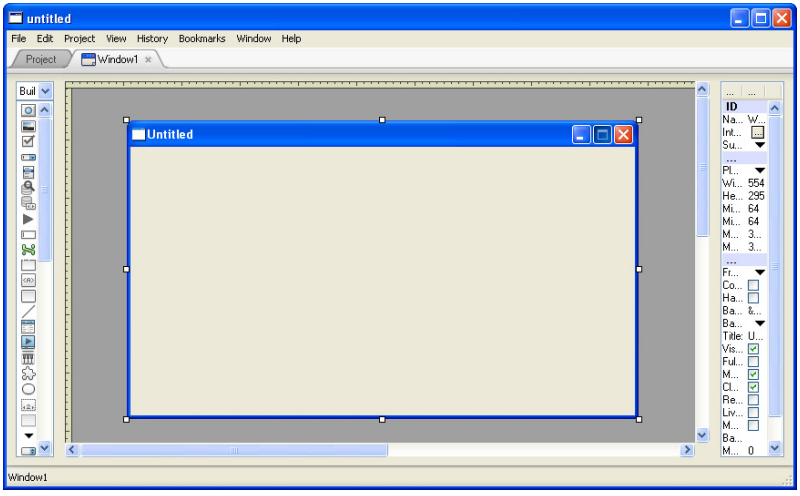
- **Build Application:** The Build Application item is equivalent to the Build button in the Main Toolbar. It builds your application according to the current Build Settings and App object's properties. For information on setting the App object's properties, see the section "Customizing the Standalone Application's Properties" on page 550.

The View Menu

The View menu has IDE configuration options that enable you to show, hide, and customize the toolbars and maximize the area of the window used by the current editor:

- **Editor Only:** If the Editor Only menu item is selected (has a checkmark to its left), the area of the window devoted to the current editor is maximized, hiding the toolbars and minimizing any panes to the left and right of the editor area. For all editors, it hides the Main and Editor toolbars and the Bookmarks bar. It also minimizes the Properties pane and Controls pane, if appropriate for that editor. Figure 23 shows a Window Editor in its maximized state. For a Code Editor, the Editor Only command reduces the Browser area to a little sliver, like the Controls pane in Figure 23. If Editor Only is selected, you can revert to the "normal" view by deselecting this menu item. If you have resized the panes prior to selecting Editor Only, REALbasic remembers your settings when you deselect Editor Only.

Figure 23. A Window Editor in Editor Only mode.



- **Show Code/Show Layout:** Available only for Window and Container Control Editors. If the Layout editor for a window or container control is shown, the Show Code menu item switches to its Code Editor; if the Code Editor is shown, the Show Layout menu item switches to the Layout Editor view. These menu items are the equivalents of the Code Editor and Window Editor icons on the left side of the Window Editor and Code Editor toolbars.

- **Control Order:** Available only for Window and Container Control Layout Editor views. If the Control Order menu item is selected (has a CheckBox to its left), the control order for the controls in the window are shown as numbered badges. To hide this information, deselect the Control Order menu item.
- **Menu Layout:** Available only for a Menu Editor. It has submenu items for previewing the current menubar on any of the four platforms on which you can build the application: Windows, Mac OS X, Mac "classic", or Linux. These submenu items are equivalent to the four operating system icons on the left side of the Menu Editor toolbar.
- **Main Toolbar:** The Main Toolbar menu item has a submenu for hiding and customizing the Main Toolbar. The Customize submenu item displays a "mover" dialog box that you can use to add or subtract items from the Main Toolbar.
- **Bookmarks Bar:** The Bookmarks Bar submenu has items for hiding and customizing the Bookmarks bar. This is the row of local bookmarks that you have added to the current project via the Add Bookmark button or the Bookmarks ▶ Add Bookmark menu item. The Customize submenu item displays a dialog box in which you can edit the text and location of the bookmarked items.
- **Editor Toolbar:** The Editor Toolbar is just below the row of tabs and has buttons for adding items to the object being edited. Each editor has its own toolbar, so this menu item pertains to the current editor's toolbar. The Editor Toolbar menu item has a submenu for hiding and customizing the current Editor Toolbar.

The History Menu

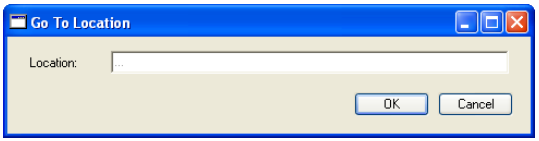
The History menu keeps track of the screens that you use as you work on your project. As you work, it automatically adds the names of the screens to the menu. You can go directly to any screen that you have previously visited by choosing its name from the History menu.

The History menu also includes menu commands for the navigation-related items in the Toolbar:

- **Backward:** Moves to the previously-viewed screen. This is equivalent to the Back button in the Toolbar.
- **Forward:** Moves to the next screen that you viewed (assuming that you have moved backwards in the list of viewed screens). This is equivalent to the Forward button in the Toolbar.
- **Home:** Moves to the Project Editor, the screen that you see when you double-click the REALbasic IDE application or choose File ▶ New Project. This is equivalent to the optional Home button that can be added to the Main Toolbar.
- **Go to Location:** Allows you to enter the name of an Editor or an item. This is equivalent to entering the item name into the Location area in the Main Toolbar. If the Location area is shown when this menu item is selected, it moves the text insertion point into the Location area. Otherwise, it opens a dialog box or sheet

window into which you can enter the location. Figure 24 shows the Go To Location dialog box that is displayed only when the Main toolbar is hidden.

Figure 24. The Go To Location dialog box.



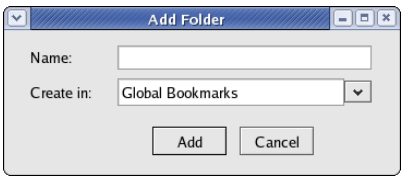
The Bookmarks Menu

The Bookmarks menu contains commands for bookmarking individual items in your projects. As you add bookmarks, the names of the bookmarked items are added to the Bookmarks menu. Go to a bookmarked item by choosing its name from the Bookmarks menu or clicking it in the Bookmarks bar.

You can bookmark items from several projects and they will be available in all open REALbasic IDE windows. The syntax that REALbasic uses is *projectname.editorname*, that is, the name of the project on disk, followed by a dot, followed by the name of the editor. If the item is a control in a window, the dot syntax is extended: *projectname.editorname.controlname.* If the project name is omitted, the current project is assumed. Use the Global Bookmarks option when you want the bookmark to be available in all of your projects. This makes it easy to navigate to an item in another project.

- **Show all Bookmarks:** Opens a dialog box that lists all the global bookmarks available. You can edit the names and/or locations of your global bookmarks from this dialog box. It works the same way as the Edit Bookmarks dialog box for your local bookmarks in the Bookmarks bar. For information on editing bookmarks, see the section "The Bookmarks Bar" on page 40.
- **Add Bookmark:** Adds the current item to either the Bookmarks menu or the current Bookmarks bar. If you have added a Bookmark Folder to either the menu or the bar, you can also add the current item to any folder. For information on adding a bookmark, see the section "The Bookmarks Bar" on page 40.
- **Add Bookmark Folder:** Displays a dialog box that enables you to name and add a folder to either the Bookmarks menu. When you choose Add Bookmark Folder, the following dialog box appears:

Figure 25. The Add Bookmark Folder dialog box.



After you have created a bookmark folder, the Add Bookmark dialog's Create In drop-down list offers to add the new bookmark to the folder.

The Window Menu

The Window menu has items for managing the REALbasic IDE window. You can have more than one window open for the same project and you can have multiple projects open. It also has items for selecting among tabs in the current window.

- **Minimize:** Minimizes the IDE window to the Taskbar (Windows and Linux) or the Dock (Mac OS X). This is equivalent to clicking the Minimize button in the window's Title bar. When the IDE window is minimized, you can click on its name or icon to restore the window to its previous size and position.
- **Maximize:** Maximizes the IDE window to fill the screen. This is equivalent to clicking the Maximize button in the IDE window's Title bar. When the window is maximized, this menu item changes to Restore. Choose Restore to return the IDE window to its previous size and position.
- **Next Tab:** Next Tab is equivalent to clicking on the tab to the right of the current tab in the Tab bar.
- **Previous Tab:** Previous Tab is equivalent to clicking on the tab to the left of the current tab in the Tab bar.

The Help Menu

The Help menu provides easy access to the built-in REALbasic documentation as well as online information from the REALbasic web site. The Help menu has the following menu items:

- **Getting Started:** Opens the REALbasic QuickStart using Adobe Acrobat Reader. If the QuickStart file is not installed, this menu command displays a dialog box enabling you to download it from the REAL Software web site.
- **Tutorial:** Opens the REALbasic Tutorial in a new window using Adobe Acrobat Reader. If the Tutorial file is not installed, this menu command displays a dialog box enabling you to download it from the REAL Software web site.
- **User's Guide:** Opens the REALbasic User's Guide in a new window using Adobe Acrobat Reader. If the User's Guide file is not installed, this menu command displays a dialog box enabling you to download it from the REAL Software web site.
- **Language Reference**: Opens the Language Reference. Use it to read descriptions of REALbasic objects, check syntax, and use code examples. For more information about the Language Reference, see the section "Using the On-Line Help" on page 21.
- **REALbasic on the Web:** Opens your default web browser application to REAL Software's home page on the web.
- **REALbasic Feedback:** Opens the REALbasic Feedback page, where you can enter bug reports and feature requests and search the Feedback database by keyword.

- **About REALbasic:** Opens a window that provides information on the version of REALbasic that is running and identifies the licensed user. On Mac OS X, the About REALbasic menu item is in the REALbasic 2005 menu.
- **Register:** Opens a dialog box that allows you to enter your license code into a demo version of REALbasic.

Working with Projects

All of the windows, menus, modules, pictures, sounds, QuickTime movies, plug-ins, databases, and programming code that make up an application are stored in a Project document. Projects give you a convenient way to organize the objects that make up your application. You can have several projects open at once and you can have several windows open per project.

Projects can contain any of the following items:

- Windows
- Menu bars
- Classes
- Class interfaces
- Modules
- File Type sets
- Container Controls
- Pictures
- Sounds
- QuickTime™ movies
- Databases
- AppleScripts (Mac OS only)
- Resource files (Mac OS only)
- Cursor resources
- Folders
- Documents of other types, which are treated as text strings

Each item is denoted by an icon that denotes its type and, in some cases, its subtype. For example, different window types have their own icon.

If some of these items are not familiar to you, don't worry. You will learn more about them in later chapters.

Double-clicking on an item in the Project Editor will either display the item in its editor or a viewer for the item, if REALbasic has no editor for that type of item.

NOTE: There is one exception to this rule. You can encrypt an item in the Project Editor to prevent others from accessing the item and any code associated with it. An encrypted item functions normally in the built application, but it cannot be viewed or edited within the REALbasic IDE. Encrypted items have a small key in the lower-right corner of its icon. When you double-click an encrypted item, REALbasic displays an alert that tells you that the item is encrypted. For more information, see "Encrypting Your Source Code" on page 246.

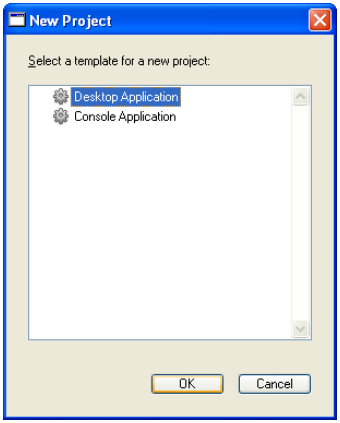
Creating A New Project

When you open REALbasic by double-clicking on the REALbasic application icon, a new project is created for you automatically. By default, it is a Desktop Application project, as described in the next section. However, you can customize the default project that opens automatically when you start REALbasic. For more information, see the section "Creating Project Templates" on page 59.

If you immediately resize the IDE window after creating a new project, REALbasic will remember this size and use it as the default IDE window size for subsequent new projects.

If you have a project open and wish to begin a new one, simply choose File ▶ New Project. REALbasic presents the New Project dialog box. It enables you to choose the type of project template that you want to base your new project on. By default, it contains two items.

Figure 26. The New Project dialog box.



The default items are:

- **Desktop Application:** This is the default template for any standard REALbasic project that uses any graphical user interface (GUI). All REALbasic Desktop Application projects have a window and a menubar. The Desktop Application template includes these two items. Except where noted, the *User's Guide* deals only

with Desktop Applications. When it refers to a REALbasic project, it means Desktop Application project.

- **Console Application:** This is the default template for a REALbasic application that runs only in the Terminal window (Mac OS X and Linux), the command line (Windows), or completely in the background. A Console Application cannot have a menubar or windows. These items are omitted from the Console Application template. Use this option only when you want to create an application that has no graphical user interface of its own. For example, a web or mail server application can be written so that it has no graphical user interface. For more information on console projects, see the entries in the *Language Reference* for the ConsoleApplication and ServiceApplication classes.

  By default, a new Desktop Application project contains three items, Window1, the main window, MenuBar1, the application's default menu bar and menus, and a special class called the App class, which you can use to manage application-wide functions of the application. For information about this class, see the section "The Application Class" on page 453. The default project for a Console Application lacks the items that are needed to build and maintain an interface. It has only one item, the App class.

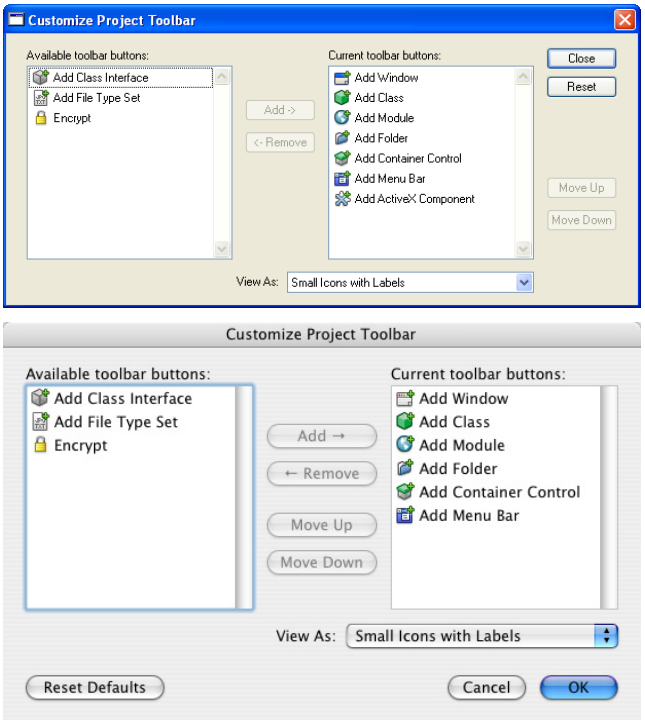
  You can add your own templates to the list presented in Figure 26. When you create a new project from a template, the Project Editor will contain all the additional objects that were saved with the project when the template project was created. For more information on adding templates, see "Creating Project Templates" on page 59.

Configuring the Project Editor Toolbar

The Project Editor Toolbar (just below the row of Tabs) has buttons for adding items to the project. By default, it has buttons for adding classes, class interfaces, modules, folders, menubars, and windows. If you like, you can modify the Project Editor toolbar with the View ▶ Editor Toolbar ▶ Customize submenu. Note that the Project pane must be selected to customize the Project Editor toolbar rather than another editor's toolbar.

The following dialog appears:

Figure 27. The Customize Project Editor Toolbar dialog box.



The Customize Project Toolbar dialog box uses a "mover" interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. Listed on the left are optional items that can be added to the toolbar. By default, only File Type Set is not included in the toolbar.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (shown in Figure 18).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down or select the item to be moved, drag it to the desired location, and drop it between two items. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down menu. Your choices are:
  - Big icons with labels.

- Small icons with labels,
- Big icons (no labels),
- Small icons (no labels),
- Labels only.

- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or the Reset Defaults button (Macintosh).

Adding Items to Your Project

The method you use to add items to a project depends on the type of item you wish to add. You add REALbasic project items such as windows, classes, class interfaces, menubars, and modules by clicking on a button in the Project Editor toolbar (just below the row of tabs) or with the Project ▶ Add submenu.

If you have a picture, sound, movie, or REAL database you wish to use in your project, you can add with the File ▶ Import menu item. It displays an import-file dialog box that enables you to navigate to the item to be imported. Choose the item and click the Import button.

You can also import items by dragging the file from the desktop and dropping it into the Project Editor. If REALbasic stores a shortcut (alias on Macintosh) to an external item, it is displayed in italics within the Project Editor. You will learn in later chapters how to add each type of item that can appear in the Project Editor.

Organizing Project Items

If your application has many items in its Project pane, you may want to organize them rather than leaving them in the order in which they were created. First, you can reorder an item by holding the mouse button down on the item and dragging it up or down. As you drag, you can drop it to a new position. The new position is indicated by a horizontal line between existing items as you move the dragged item over it. Release the mouse button to drop the item into its new position.

You can also group items by adding a folder to the Project pane and storing some items in the folder. For example, you might want to create a folder to hold all the movies and another folder for all the sounds. First, create the folder by clicking the Add Folder button in the Project Editor toolbar or choose Project ▶ Add ▶ Folder. A folder has one property, its name. Use the Properties pane to give the new folder a meaningful name. Then drag items that you wish to group together into the folder. To drag into a closed folder, drag in a diagonal direction to the right to indicate that the item should be placed in the folder instead of above or below. If you want to add new items to a folder, open the folder and then click the Add button in the toolbar for the desired type of item.

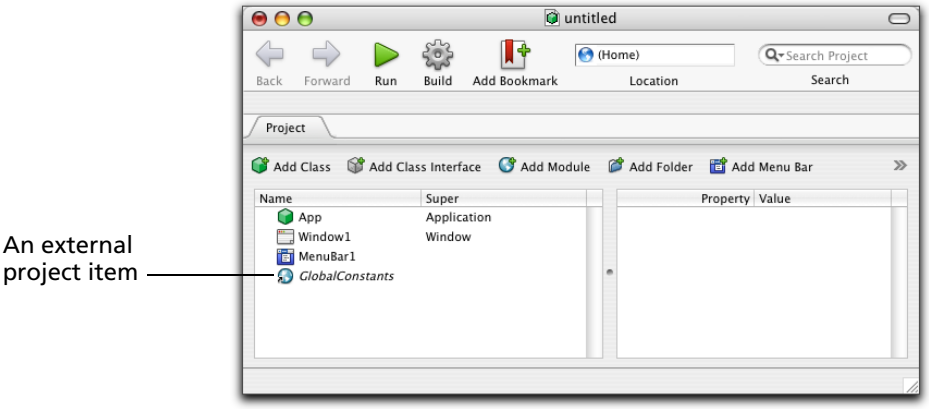
Note that your use of these two features does not affect the functionality of the project. They are available as a convenience to you, making it easier to work with projects with a lot of items.

External Project Items

It is also possible to include windows, classes, or modules in a project that are actually stored in external files. This feature allows more than one project to use the externally stored item. When you modify an external project item in REALbasic and then save your project, your changes are written out to the external file on disk. When you open any other project that refers to the same external file, that project will reflect your changes. However, REALbasic does not allow you to open more than one copy of REALbasic accessing the same external project item simultaneously.

To add an item stored on the desktop to a project as an external item, hold down the (Ctrl and Shift keys (⌘ and Option keys on Macintosh) while you drag the item from the desktop to the Project Editor. You'll know you've done it correctly because the icon in the Project Editor will have a small shortcut (alias on Macintosh) badge and the name will be in italics, just like shortcuts on the desktop. In Figure 28 an externally stored module has been added to the project.

Figure 28. A module as an external project.



You can add an item to another project as either an external item or a regular item. To add it as an external item, hold down the Ctrl and Shift keys (⌘ and Option keys on Macintosh) when dragging to the Project Editor.

When you save a project that contains external items, REALbasic will display a Save As dialog box for each external item, followed by the one for the project itself. This gives you an opportunity to save your modified external project items to a new location, leaving the original ones intact. External project items that you haven't modified are not presented in this way.

If an external project item is set to Read Only in Windows or Linux or locked in the Finder (on Macintosh), you can't modify that item within the REALbasic IDE, though you can still view it. This provides a convenient way to protect external items (which may be shared by many projects) from accidental modification. It also provides a way to use REALbasic with some version control systems, as long as these systems can use locking at the OS level to reflect the checked in/out state of each file. (Note that if an external file is changed on disk by something other than

REALbasic — such as a version control system — you'll need to re-open the project to reload that item.)

In addition to REALbasic items that can be stored either internally or externally, several other types of items are stored as external items by default. These include movies, sounds, and pictures.

Removing Items from Your Project

You can remove items from a Project by clicking once on the item in the Project Editor to select it and then pressing the Delete key. You can also select the item and choose Edit ▶ Cut or Edit ▶ Delete. You can also use the Delete command in the Project Editor's contextual menu, described in the following section.

The Project Editor Contextual Menu

The Project Editor has its own contextual menu that makes it easy to perform actions directly from the Project Editor. Right+click on a project item (Control-click on Macintosh) or on the Project Editor itself to display a contextual menu. The items on the menu vary depending on object type and may include some or all of the following items:

- **Add:** Add has a submenu that offers to add any type of REALbasic object that can be added to the Project Editor. This includes windows, classes, class interfaces, container controls, modules, folders, menu bars, file type sets, and REAL SQL Databases. This menu item duplicates the functionality of the Project ▶ Add menu item.
- **Edit:** Opens the item for editing within REALbasic. For example, if the item is a module, it opens the Code Editor for the module. If the item is a movie, it opens the movie using the default media player for the movie type, if available. You can also open files by simply double-clicking the item in the Project Editor. (If the object is encrypted, the Edit command is unavailable).
- **Edit Window:** Opens the window for editing in a Window Editor. This item is available only if the item is a window.
- **Edit Source Code:** Opens the Code Editor for the window. This item is available only if the item is a window.
- **Delete:** Removes the item from the project. You can undo this action with the Edit ▶ Undo Delete menu command.
- **View:** Displays the item within REALbasic but does not give you the ability to edit the item. For example, if the item is a picture, View displays the image in a new window. If the item is not editable within REALbasic, you can also view it by double-clicking it.
- **Play:** Appears only if the item is a sound file. Plays the sound.
- **Open File:** Opens the file as if double-clicked from the desktop. For example, if the item is a picture, it opens the item in the application indicated by its file extension or Type and Creator codes.

- **Show on Disk:** Opens the file's folder and highlights the file. For external items only.
- **File Path:** Displays a hierarchical menu to the external item, allowing you to open any folder enclosing the file.
- **Encrypt:** Displays the Encrypt… dialog box, allowing you to enter an encryption password and encrypt the object. This feature is most typically used to hide code in windows, classes, or modules that you distribute or sell to other developers. For an example, see the section "Encrypting Modules" on page 318. Available only if the item is not encrypted. You can't encrypt items that originate outside of REALbasic, such as movies, sounds, and images.
- **Decrypt:** Displays the Decrypt… dialog box, allowing you to enter the encryption password to decrypt the object. An encrypted object in the Project Editor has a small key in the bottom-right corner of its icon. Available only if the item is encrypted.
- **New:** Appears for the window's contextual menu and as the last item of a project item's contextual menu. Use it to add a new item to the project. This command duplicates the functionality of the New item in the File menu. Use it to add a new window, class, class interface, module, file type set, folder (for organizing project items), menubar, ActiveX component, or database.
- **New Subclass:** Appears only when the selected item in the Project Editor is a class or subclass. Creates a new subclass, adds it to the Project Editor, and automatically sets its super class to the selected class, that is, it creates a subclass of the selected class. The new subclass is automatically named Custom*ClassName*, where *ClassName* is the name of the super class. After creating the subclass, you can use the Properties pane to make any necessary modifications to the new subclass, including changing its Super class. See Chapter 9, "Creating Reusable Objects with Classes" on page 419 for information on classes and subclasses.
- **Implement Interface:** Opens the Implement Interface dialog box. This enables you to implement a class interface for the selected item. Available only for windows and classes. It contains a drop-down list of all the built-in and custom class interfaces in the project. Choose among the types of interfaces presented in the drop-down list. When you specify a class interface via this dialog box, REALbasic will automatically add all the method declarations specified by the class interface to the item's Method Editor and will open the Method Editor to the first such method. A window or class can have more than one class interface. Repeat this process to specify additional class interfaces for the item. For more information on interfaces, see the section "Class Interfaces" on page 458.
- **Extract Interface** Opens the Extract Interface dialog box. Available only for windows and classes. This enables you to create a new class interface that uses some of the methods of the current item in the new class interface and makes the current class or window an implementer of the new class interface.

- **Extract Superclass** Opens the Extract Superclass dialog box. Available only for classes that do not have a superclass. This enables you to create a new superclass from the current class. You can give the new superclass a name and select some or all of the methods and properties of the current class to be methods and properties of the superclass instead. When you accept this dialog, the new superclass is added to the project and any methods or properties that you selected become methods of the superclass and are removed from the current class.
- **Export...**: Used to export the item to disk. When you choose Export, a Save-file dialog appears, enabling you to save the item to disk. You can import the item into another project by dragging it from the desktop to the Project Editor or importing it as an external item (in which the item remains on disk and can be shared among projects) by holding down the Ctrl and Shift keys (⌘ and Option keys on Macintosh) while you drag the item from the desktop to the Project Editor.

Saving Your Project

When you want to save the changes you have made to your project, choose File ▶ Save. If you are making lots of changes, save your project often just as you would if you were editing a document in a word processor. If you aren't sure whether you want to keep the changes you have made, you can choose not to save your project or choose Save As from the File menu and save the project under another name. This will keep your original project intact.

As noted above, you can save external project items independently of the project via the items' contextual menus.

Saving as XML

Projects can be saved in XML format using the File ▶ Save or File ▶ Save As commands and then choosing XML from the Save as Type (Windows) or Format (Macintosh) pop-up menu. When you save in XML format, you are really doing an export, not a save; the project stays associated with its original file location and format.

Opening XML

Projects can be imported from XML using the File ▶ Open command. Also, all text files will appear in the Open dialog, but only those that contain proper XML data can actually be read. If the read is successful, a new project called "SomeXMLFile (Converted)" will be created, where "SomeXMLFile" was the name of the XML file.

Creating Project Templates

If you have several items you commonly use in every project, you can save them in a project file and make the project file a template for new projects. When you create a new project based on the template, REALbasic creates a new untitled project that is an exact copy of the template. The template project itself remains unchanged. This lets you create a new project using existing project items without worrying about modifying the original items.

The most efficient way to use a template is to place it in a special directory in the same directory as REALbasic called "Project Templates". If you do so, your list of templates will be presented whenever you choose File ▶ New Project.

If you want REALbasic to use a particular custom template automatically, name it "Default New Project" and place this template in your Project Templates folder. This template will be used when REALbasic is launched.

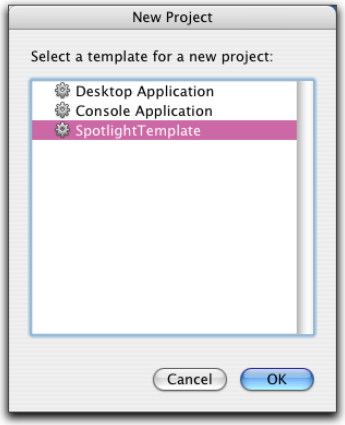
To use a template to start a new project, do this:

1 **Copy the projects you wish to use as templates to the "Project Templates" folder in the folder that contains the REALbasic application.**

2 **Launch REALbasic and choose File ▸ New Project.**

REALbasic presents the New Project dialog box. It lists all the template projects in the Project Templates directory and also offers the option of creating either of two types of blank projects.

Figure 29. The New Project dialog box with a custom template.



3 **Highlight the desired template and click OK.**

REALbasic will use that item as the basis of the new project. It will open as "Untitled" but will have all the classes, modules, windows, and so on from the template file. Changes to a new project based on a template will not affect the template.

CHAPTER 3

Building a User Interface

Your application's user interface is probably the most important part of any application. The old saying "You don't get a second chance to make a first impression" couldn't be more true when it comes to your application's user interface. If the interface is unintuitive and sloppy, the user will react the same way they might react to someone who has poor communication skills and cares little for his appearance. Using your application will be frustrating at best and, at worst, the user will give up and look for another solution to his problem. This leaves you with whatever goals you had for your application unfulfilled.

Fortunately, REALbasic makes building your application's user interface so fast and easy that you can spend the time you need to get the interface just right. REALbasic's built-in Interface Assistant™ actually helps you build a proper, clean interface.

In this chapter you will learn just about everything you need to know about creating all of the elements that make up your application's user interface. You will learn some guidelines to follow when creating your interface and how to build windows and menus.

Contents

Working with Windows

Typically, most of an application's user interface will be in the application's windows. This, of course, is highly application-specific. Some applications have no windows at all. REALbasic makes it easy to create new windows of just about any type. You create your user interface by creating its windows and adding interface controls such as PushButtons and CheckBoxes.

By default, a REALbasic Desktop Application project has one window which is displayed automatically when the application runs. Typically, you will begin designing your application's interface by adding controls to this window and enabling the controls by writing code.

To add additional windows to an application, you follow the following procedure:

- Add a new window to the project by clicking the Add Window button in the Project Editor toolbar or with the Project ▶ Add ▶ Window command,
- Set the window's Frame type and other properties using its Properties pane,
- Add controls to the window,
- Add code as needed,
- Add code to display the window in the finished application (see the section "Opening Windows" on page 250).

The following sections review the eleven types of windows supported by REALbasic. In addition to these window types, you can design message dialog boxes without formally creating a window for the message. You do so using the MessageDialog class, which is described in the section "The MessageDialog Class" on page 80.

Window Types

REALbasic supports eleven types of windows. The window type is set by its Frame property. Some types, however, are rarely used in modern applications and are retained only for historical reasons. A few specialized windows are supported on Mac OS X only. The types of windows are:

- Document
- Movable Modal dialog
- Modal dialog
- Floating

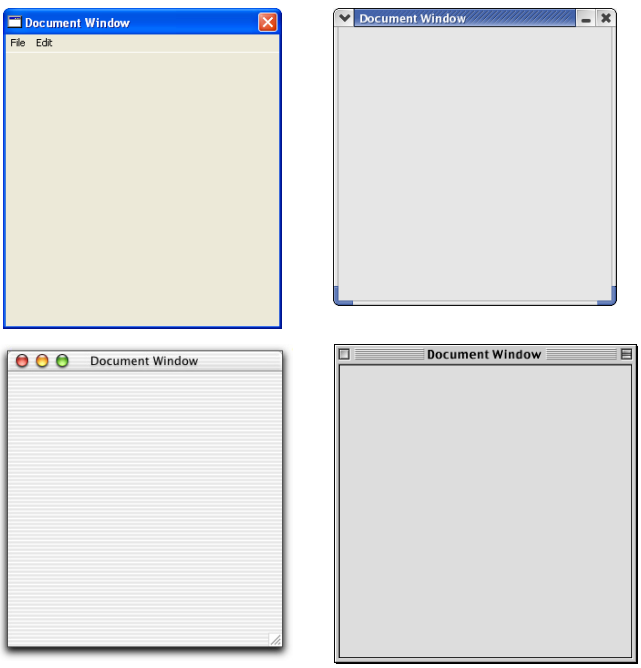
- Plain box
- Shadowed box
- Rounded (functionality available only on Mac OS “classic”)
- Global Floating
- Sheet window (functionality available only on Mac OS X)
- Metal window (functionality available only on Mac OS X 10.2 and above)
- Drawer window (functionality available only on Mac OS X 10.2 and above)

The type you choose for a particular window depends mostly on how the window will be used.

Document

The Document window is the most common type of window. When you add a new window to a project, this is the default window type. It is also the window type for the default window, Window1. Document windows are most often used when the window should stay open until the user dismisses it by clicking its close box (if it has one) or clicking a button programmed to close the window. The user can click on other windows to bring them to the foreground, moving the document window behind the others. Figure 30 on page 63 shows an example of a small, blank document window. When you first launch REALbasic, the default Desktop Application project includes one blank Document window.

Figure 30. Document windows.



Document windows can have a close box, a maximize box, and a grow handle (making them user-resizable). Mac OS X Document windows have the standard set of red, yellow, and green buttons in the Title bar.

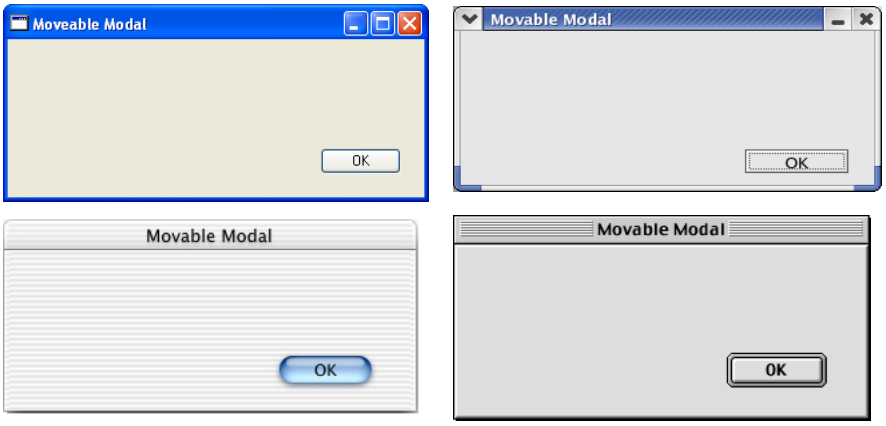
On Windows and Linux, the default menubar, Menubar1, appears in the window by default. You can choose to display the window with no menubar by setting the MenuBar property of the window to None, or, if you have created additional menubars, choose a different menubar.

When you add a new window to your project, it defaults to the Document Window type. You can modify its type by setting the Frame property (see "Using a Window's Properties Pane" on page 71).

Movable Modal

This type of window stays in front of the application's other open windows until it is closed. Use a Movable Modal window when you need to briefly communicate with the user without the user having access to the rest of the application. Because the window is movable, the user will be able to drag the window to another location in case they need to see information in other windows in order to finish what they are doing in the Movable Modal window. Figure 31 shows examples of a blank Movable Modal window on each operating system.

Figure 31. Movable Modal windows.



On Windows and Linux, a Movable Modal window has minimize, maximize, and close buttons in the Title bar. In Windows MDI interfaces, the window opens in the center area of the screen rather than in the center area of the MDI window. Therefore, the Movable Modal window may open outside the MDI window.

On Macintosh, Movable Modal windows cannot have a close box, so you need to include a button that the user can click to dismiss the window unless the window will dismiss itself after the application finishes a particular task. Also, Macintosh Movable Modal windows are not resizable by the user and cannot have maximize box. This means you will have to consider the amount of available screen space the user will have in determining the size you will make a Movable Modal window.

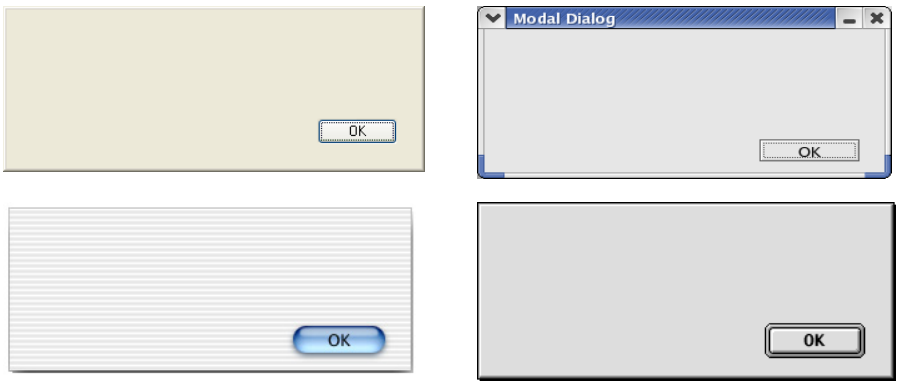
NOTE: There is one exception to the rule regarding Movable Modal windows being in front of all other windows. If a Movable Modal window or one of its controls executes code that opens a Floating window, the Floating window will be in front of the Movable Modal window. However, it is poor interface design for a Movable Modal window to open another window because Movable Modal windows are mostly used in situations where the interaction with the user will be brief.

Modal Dialog

These windows are very similar to Movable Modal windows. The only difference is that Modal Dialog windows have no Title bar, so they cannot be moved. On Windows, a Modal dialog box has no minimize, maximize, or close buttons. In Windows MDI applications, a Modal Dialog window opens in the center area of the screen rather than the center area of the MDI window. Therefore, a Modal Dialog box may open outside of the application's MDI window. On Linux, Modal Dialogs are modal but have a Title bar and close and minimize buttons.

The Page Setup dialog box is an example of a Modal Dialog window.

Figure 32. A Modal Dialog window.



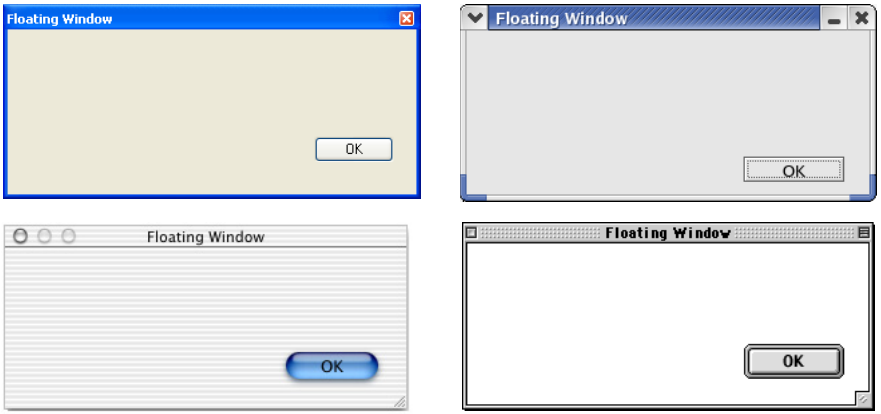
NOTE: Because Modal Dialog windows and Movable Modal windows are both modal, the same exception applies regarding floating windows opening in front of Modal windows. See the note for Movable Modal windows on page 65.

Floating

Like Movable Modal and Modal Dialog windows, a Floating window (also known as a *Windoid*) stays in front of all other windows. The difference is that the user can still click on other windows to access them. If you have more than one Floating window open, clicking on another Floating window will bring that window to the front, but all open Floating windows will be in front of all non-floating windows. Because they are always in front of other types of windows, their size should be kept to a minimum or they will quickly get in the user's way. This type of window is most commonly used to provide tools the user will frequently access.

A Global Floating Window is a Floating window that can float in front of a particular application's window or all applications' windows. They are described in the section "Global Floating" on page 68.

Figure 33. Floating windows.



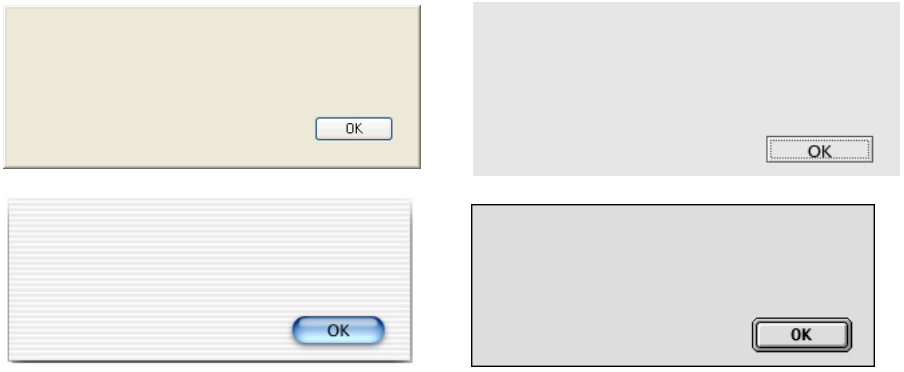
Like Document windows, Floating windows can have a close box and can be user-resizable. On Linux, Floating windows have minimize and maximize widgets.

In Windows MDI applications, a Floating window can float outside the application's own window. By default, a Windows MDI Floating window opens in the top-left area of the screen, regardless of the location of the MDI window.

Plain Box

These windows function as Modal Dialog windows. The only real difference is their appearance, as you can see in Figure 34 on page 66. Plain Box windows are commonly used for About Box windows and for applications that need to hide the desktop.

Figure 34. Plain Box windows.



On Windows MDI applications, a Plain Box window opens in the center area of the screen rather than the center area of the MDI window. Therefore, a Plain Box window may open outside the MDI window.

Shadowed Box

Like Plain Box windows, Shadowed Box windows function as Modal Dialog windows. The only difference is their appearance, as you can see in Figure 35. Shadowed Box windows are commonly used for About Box windows.

Figure 35. Shadowed Box windows.

On Windows MDI applications, a Shadowed Box window opens in the center area of the screen rather than the center area of the MDI window. Therefore, a Shadowed Box window may open outside the MDI window.

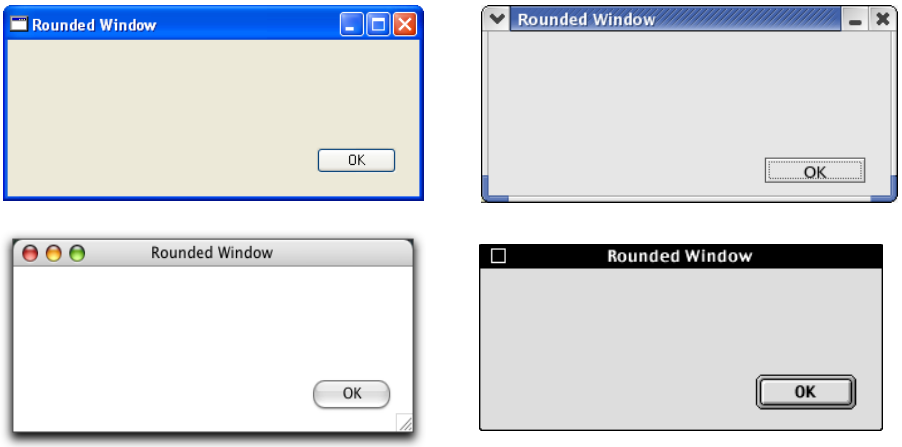
On Mac OS X, a Shadowed Box window works like a Modal Dialog box with a minimize button.

Rounded

Rounded windows act like Document windows. The only differences are appearance (as you can see in Figure 36) and the fact that, on Macintosh, Rounded windows cannot have a zoom box or be resizable. On Windows, a "Rounded" window has the standard minimize, maximize, and close buttons in the Title bar and the corners are not actually rounded.

Rounded windows are not commonly used any more and there is really no reason to use them instead of Document windows. Rounded windows appear as Document windows in REALbasic Mac OS X applications.

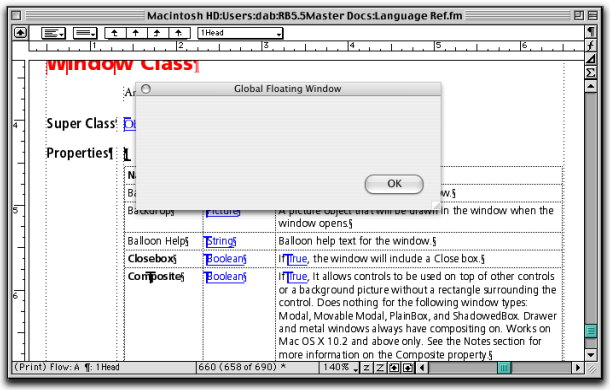
Figure 36. Rounded windows.



Global Floating

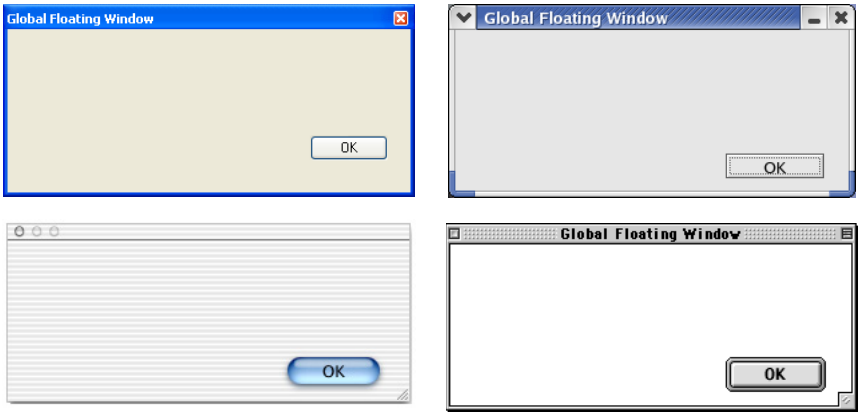
A Global Floating window looks like a Floating window, except that it is able to float in front of other applications' windows, even when you bring another application window to the front. In Figure 37 on page 68, shows a FrameMaker document window that holds the REALbasic *Language Reference*, making it the active window, yet the REALbasic Global Floating window is still in front of FrameMaker.

Figure 37. A Global Floating window in front of a FrameMaker document.



This doesn't work for a Floating window. A 'regular' Floating window floats only in front of its own application's windows.

Figure 38. Global Floating windows.

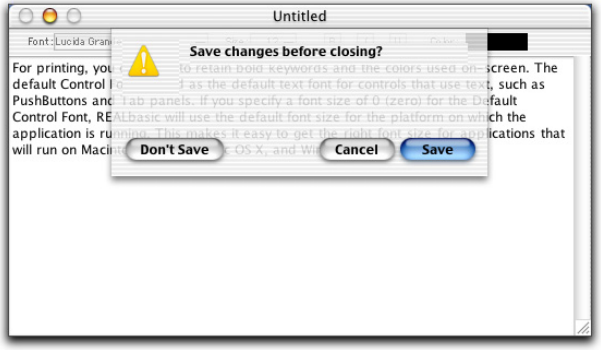


On Windows MDI applications, a Global Floating window can float outside of the MDI window. By default, it opens in the top-left area of the screen.

Sheet Window

A "Sheet window" is the official name for drop-down dialog boxes that were introduced with Mac OS X. Mac OS X uses them in place of modal dialog boxes. Figure 39 illustrates an appropriate usage of a Sheet window.

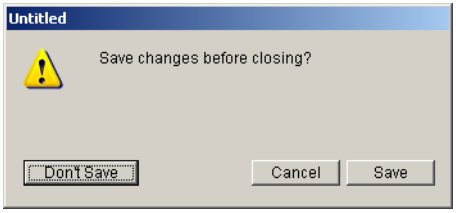
Figure 39. A Sheet window in action.



A Sheet window behaves like a Modal dialog window, except that the animation makes it appear to drop down from the parent window's Title bar. It can't be moved from that position and it puts the user interface in a modal state. The user must respond to the choices presented in the Sheet window.

Sheet windows behave as sheets only under Mac OS X. On Windows and Linux, Sheet windows behave like Movable Modal dialog windows and on Mac OS "classic" they behave like Modal dialogs.

Figure 40. A Sheet window on Windows XP.



Metal Window

A Metal window uses the metallic background that Apple uses in the "Panther" Finder and certain Mac OS X software products such as QuickTime, iTunes, and Safari. A Metal window has a Grow handle and the standard Title bar with close, minimize, and maximize buttons of a Document window. The Metal window type requires Mac OS X version 10.2 or above.

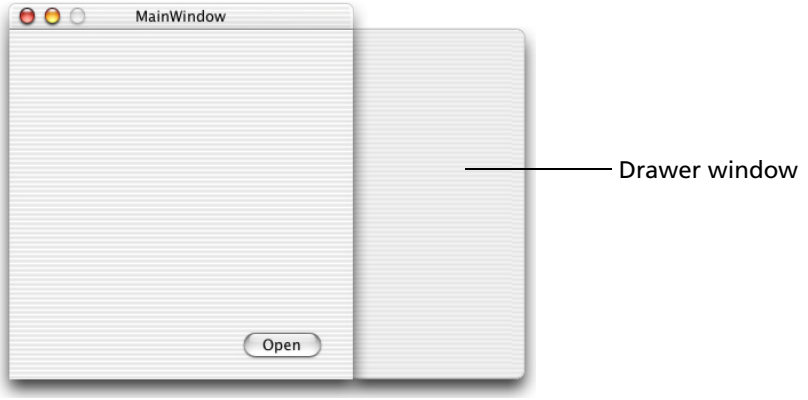
Figure 41. A blank Metal window (Mac OS X).

On Windows, Linux, and Mac OS 8-9 "Classic," a Metal window looks like a regular Document window.

Drawer Window

A Drawer window was introduced in Mac OS X. A Drawer slides out of a side or the top or bottom of a main window to provide supplemental information. For example, the Mac OS X Mail application uses a Drawer window to display the user's list of mailboxes. A Drawer window in REALbasic is supported only on Mac OS X 10.2 and above. On Windows and Linux, a window of this type appears as a separate floating window.

Figure 42. A blank Drawer window (Mac OS X).



Display a Drawer window using the ShowWithin method of the Window class. You specify the parent window from which the Drawer window slides out and you can control the border from which the window slides (either side, top, bottom, or system default, in which the system figures out where there's room for the Drawer window).

Custom Window Types

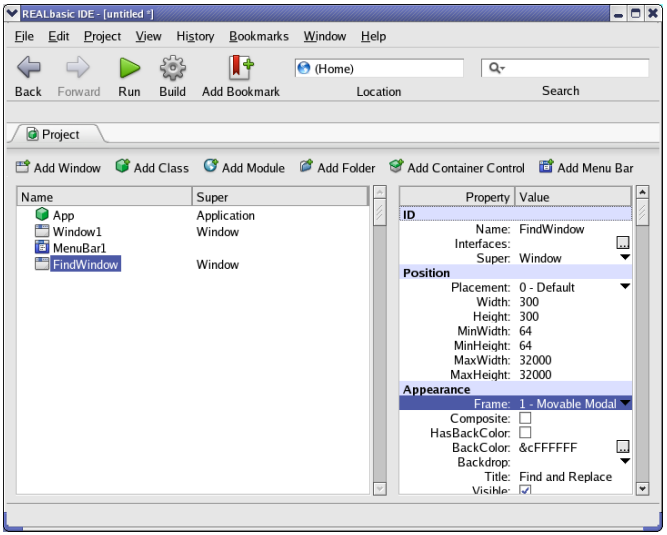
The Window class has a property, MacProcID, that allows you to create custom window types. This property gives you more options than described in this section. However, these custom types are supported only on Macintosh. All of the window types described in this section are cross-platform, with the limitations noted in each section.

For more information, see the discussion of the MacProcID property of the Window class in the *Language Reference*.

Using a Window's Properties Pane

When you click on a window's name in the Project Editor, the Properties pane changes to show the window's properties that can be set *in the development environment* — as opposed to properties that can be set via code. A selected item in the project and its properties is shown in Figure 43.

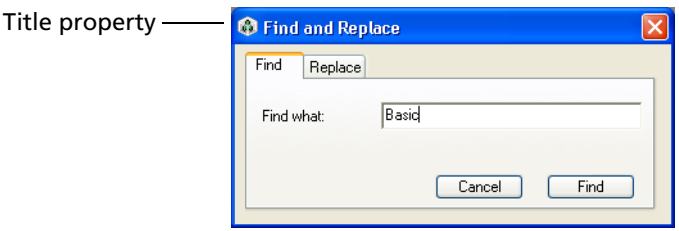
Figure 43. A Movable Modal dialog window in the project and its properties.



You change a window's properties by entering values into the enterable areas in the Properties pane, by making menu selections, and by selecting or deselecting CheckBoxes.

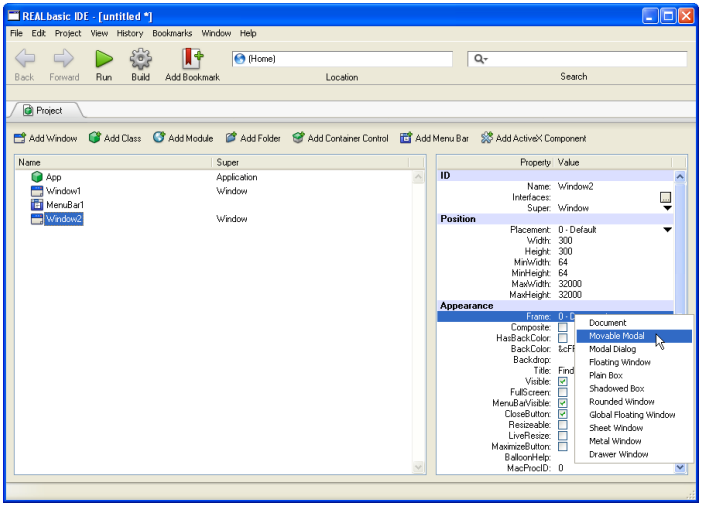
When you enter a value into an entry area, press the Return key to commit the entry. For example, this window's Title property was changed from the default text, "Untitled", to "Find and Replace". The text of the Title property appears in the window's title bar when the window appears. Figure 44 shows the window in the finished application.

Figure 44. The value of the Title property in the finished application.



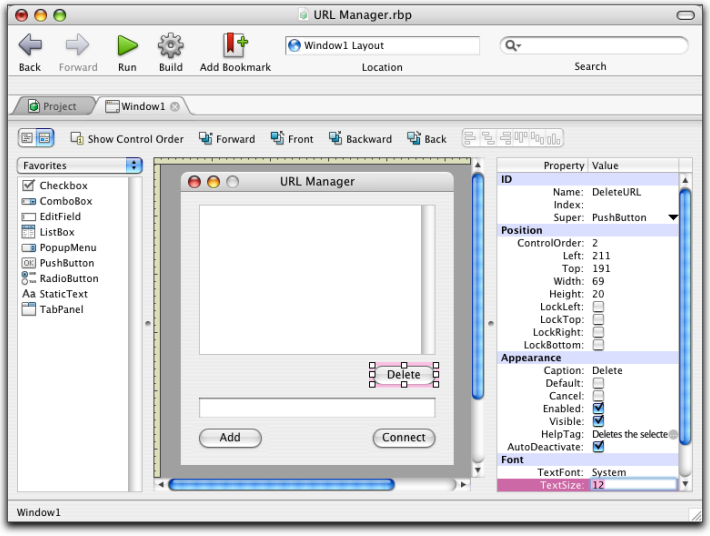
In the Properties pane, drop-down menus are denoted by the downward pointing arrow on the right side of line. For example, this window's Frame property was changed from Document to Movable Modal using the drop-down menu.

Figure 45. Changing the Frame property.



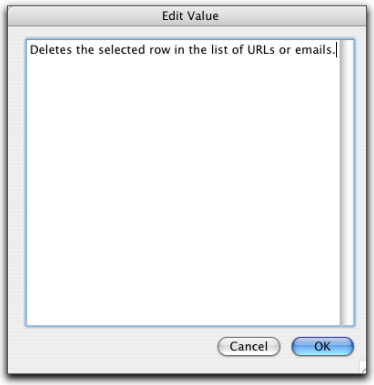
The icon with the ellipsis  (on Windows and Linux or  on Mac OS X) indicates that you can enter text by clicking the icon to display a text entry dialog. For example, the HelpTag property of the selected PushButton accepts text entry.

Figure 46. The HelpTag property with help text.



You can enter the text directly into the entry area or click the  icon to display a much larger text entry area. An example is shown in Figure 47.

Figure 47. Entering Help text in the Edit Value area.



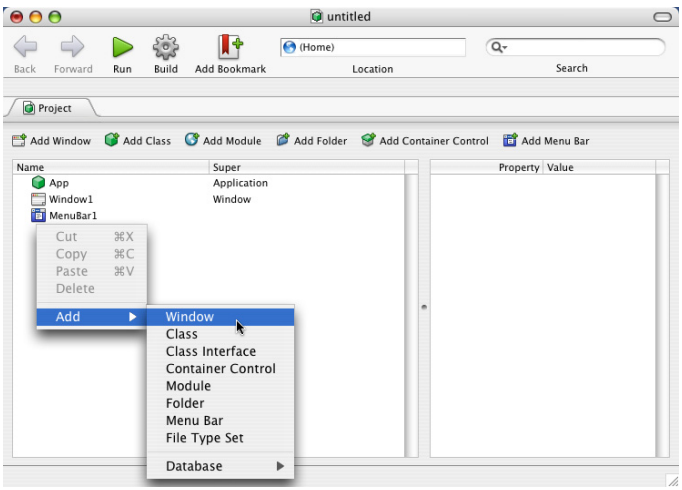
Creating Windows

When you create a new Desktop Application project, REALbasic includes a window named "Window1" to your project automatically. This is the default project window. Unless you specify otherwise, it will appear when you run your compiled application.

There are several ways to add an additional window to your project. You can click the Add Window button in the Project Editor toolbar, use a menu command, or a contextual menu.

To add a window using the menu command, choose Project ▶ Add ▶ Window. To use the contextual menu, right+click (on Windows and Linux) or Control-click (on Macintosh) and choose Add ▶ Window.

Figure 48. Adding a window to the project using the Project Editor's contextual menu.



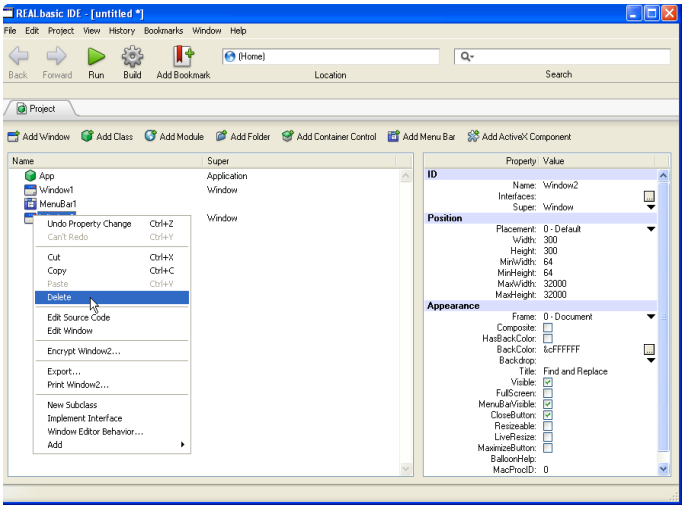
The windows in your project act as templates for windows in your application. When your application opens one of these windows, it's really opening a copy of the window. This means that your application can have several copies of the same window open at the same time. It's important to understand this when creating your user interface because there is no need to go to the extra trouble of duplicating a window in the Development environment if your application needs to open two of them at the same time.

In the REALbasic Tutorial application, the text processing application uses one window template as the document window. The File ▶ New command in the finished application allows the end-user to create as many document windows as needed from the same template. The Tutorial application needs an additional window for the Find and Replace dialog.

Removing Windows

To remove a window from your project, simply click on it once in the Project Editor to select it and press the Delete key or choose Edit ▶ Delete. You can also Right+click the window (Control-click on Macintosh) to display the contextual menu and remove it using the Delete contextual menu item.

Figure 49. Deleting a window via its contextual menu.



You can undo many actions in REALbasic. For example, if you delete a window by mistake, choose Edit ▶ Undo (Ctrl+Z or ⌘-Z on Macintosh).

Setting the Default Window

By default, the window that is included in a Desktop Application project automatically opens when your application is launched. It is called the default window. If you have created more than one window, you can make a different window the default window. Or, you can specify that no windows will open automatically when the application starts.

You set the default window using the DefaultWindow property of the App object. The App class is added to the project automatically when you create a new project.

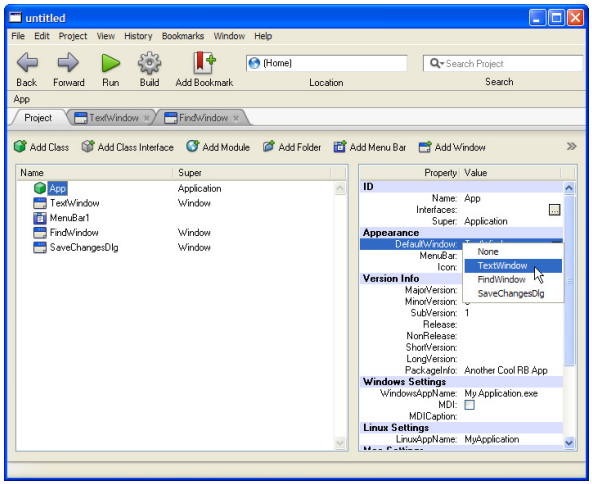
To set the default window, do this:

1 **Click on the App object in the Project Editor.**

The Properties pane changes to show the properties of the App class. The DefaultWindow property is in the Appearance group. Its pop-up menu contains all the windows in the project as well as the choice of "None."

2 **Choose the desired window from the DefaultWindow pop-up menu or, if you want no window to open, choose None.**

Figure 50. Setting the default window for the project.



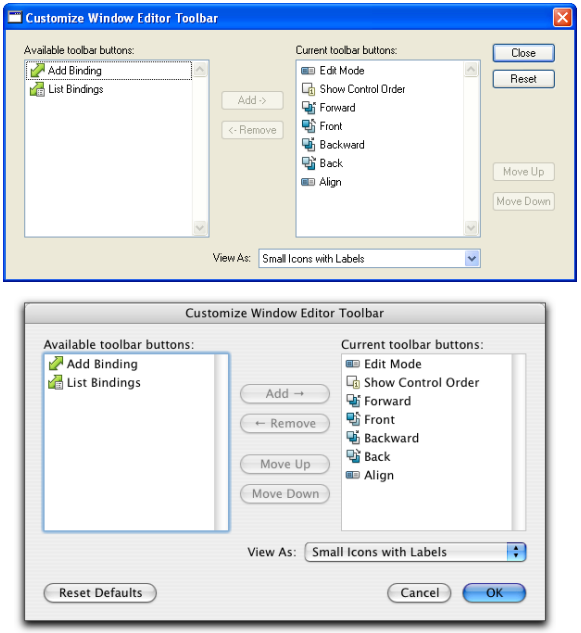
When your application launches, the default window (or no window) will open automatically.

Customizing the Window Editor Toolbar

The Window Editor toolbar has buttons for manipulating the controls in the window. By default, it has buttons for changing the control order (the order in which controls are selected when the user presses Tab) and aligning controls. To modify the Window Editor toolbar, choose View ▶ Editor Toolbar ▶ Customize submenu. (Note that a Window Editor must be selected to customize the Window Editor toolbar rather than another editor's toolbar.)

The Customize Window Editor Toolbar dialog box appears:

Figure 51. The Customize Window Editor Toolbar dialog box.



The Customize Window Editor toolbar dialog box uses a "mover" interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. The left panel contains any available items, but all Window Editor toolbar items are displayed by default.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (assuming that an item is available).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down or select an item, drag it to the desired location, and drop it between two items. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down menu. Your choices are:
  - Big icons with labels.
  - Small icons with labels,
  - Big icons (no labels),
  - Small icons (no labels),
  - Labels only.

- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or the Reset Defaults button (Macintosh).

Encrypting Windows

You can encrypt (protect) or decrypt (unprotect) a window from the Project Editor. A protected window cannot be opened in a Window Editor and no one can access any code associated with the window or any of the controls in the window. It's a good idea to encrypt any REALbasic items that you want to sell to others. They will be able to use the item in their project but cannot modify your code.

To encrypt a window, you supply a password which can be used to decrypt it later.

To encrypt a window, do this:

1 **Right+click on the window in the Project Editor (Control-click on Macintosh) and choose Encrypt from the contextual menu (see Figure 49 on page 75) or choose Edit ▶ Encrypt.**

The Encrypt *Window* dialog box appears, as shown in Figure 52.

Figure 52. The Encrypt *Window* Dialog box.

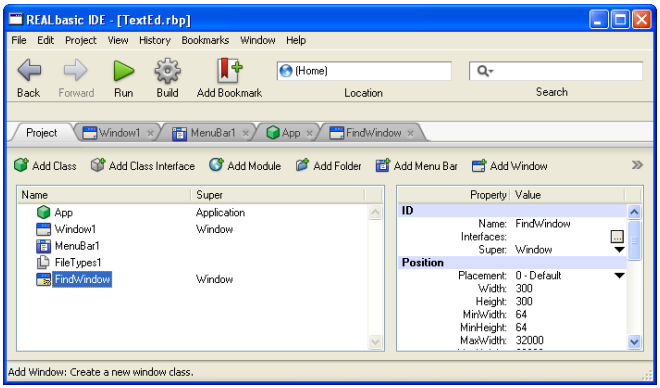
2 **Enter and confirm a password for decryption and click the Encrypt button.**

The Encrypt button will not be enabled if the "Confirm" password does not match the initial entry.

Important: Don't forget the password.

An enrypted window appears in the Project Editor with a small key in the lower right corner of the window icon. In Figure 53. the "FindDialog" window has been encrypted.

Figure 53. A project with an encrypted window.



When a user tries to open an encrypted window in a Window Editor, REALbasic displays the Decrypt *Window* dialog box, shown in Figure 54.

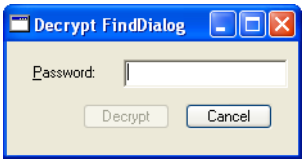
To edit the window and/or its code, you must decrypt it using its encryption password.

To decrypt an encrypted window, do this:

1 **Right+click on the window's name in the Project Editor (Control-click on Macintosh) and choose Decrypt from the contextual menu or click on the window's name and choose Edit ▶ Decrypt.**

The Decrypt *Window* dialog box appears.

Figure 54. The Decrypt *Window* dialog box.



2 **Enter the decryption password and click Decrypt.**

In a few moments, the key will disappear from the window's icon, indicating that it has been successfully decrypted. If you entered an incorrect password, a dialog box will inform you of that fact.

If you don't know the password, there is no way to decrypt the window.

Message Dialog Boxes

REALbasic offers an especially quick way to create standard message dialog boxes. There are two built in language commands that create message dialog boxes automatically. They bypass the process of creating a window, adding it to the project, and adding controls to it via the Controls pane and the Window Editor. These commands are the MsgBox function and the MessageDialog class.

These commands create only a limited range of window types. The windows are message boxes that have an icon, some text, and one or more buttons. These commands cannot be used to create dialog boxes or any other type of window that displays information in a different form. The only user input that is supported is the selection of a button to click. Also, you do not have the ability to arrange the buttons, text, and icon freely. If you need any of these features, you should create a window and add it to the project.

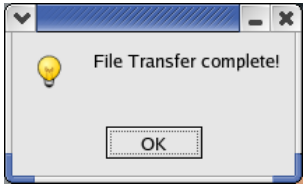
The MsgBox function

Of the two commands, the MsgBox function is the simpler one. Use it when you want to present a brief message to the user in the form of a modal message box. Simply pass the text that you want to display in the message box to the MsgBox function. For example, the following message is shown when a file upload has completed successfully:

```
MsgBox "File transfer complete!"
```

This line of code displays a Message box with the message and one button that the user can click to dismiss the window. This message box looks as shown in Figure 55.

Figure 55. The basic MsgBox message box.



The MsgBox function has two optional parameters that provide some customization. You can pass an integer that indicates that the function should display additional buttons, indicate which button is the default button, and set the icon to be displayed.

If you use the optional parameters, the MsgBox function returns an integer that tells you which button the user clicked. If you display more than the one button, you should examine the value returned by the MsgBox function.

The MessageDialog Class

Use the MessageDialog class when you need to design more complex message dialog boxes than shown in Figure 55. With the MessageDialog class, you can present up to three buttons and control their text and functionality. You can also present subordinate explanatory text below the main message.

However, since MessageDialog is a class, you cannot accomplish all of this with one line of code. You need to declare a variable as type MessageDialog, instantiate it, set its properties, and handle the result returned, which tells you which button the user pressed.

Before using the MessageDialog class, its best to read about REALbasic classes and how they work. Please refer to Chapter 9 for this information. After that, read the section on the MessageDialog class in the *Language Reference*.

The following discussion gives you an overview of the features that the MessageDialog class offers.

A MessageDialog can have up to three buttons, which are based on the ActionButton, CancelButton, and AlternateActionButton classes. They have the following properties:

Property	Description
Caption	The text displayed in the button.
Visible	Set to True to show the button.
Default	Set to True to highlight the button as the default button in the MessageDialog. By default, the ActionButton's Default property is True.
Cancel	Set to True to indicate that the button will respond to the Escape key. On Macintosh, it will also respond to the Command–. sequence. Only one button in a dialog can have the its Cancel property set to True. Some operating systems do not permit one button to be both the Default button and the Cancel button.

By default, only the ActionButton is shown, but you can show the others simply by setting their Visible properties to True.

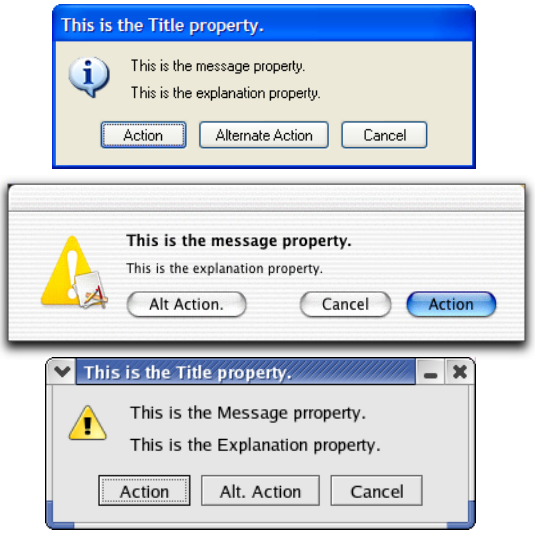
In addition, you can set the text of the message, the subordinate explanation, the type of icon shown in the dialog (no icon, Note, Warning, Stop, or Question), and the title. Not all of the icons are presented in Mac OS X. The following table shows how the icons appear on each platform and value of the Icon property.

Platform	Value of the Icon Property			
	0 (Note)	1 (Caution)	2 (Stop)	3 (Question)
Windows				
Mac OS X				
Linux				

You present the customized alert by calling the ShowModal method of the MessageDialog class.

The positions of the three buttons and the Message and Explanation properties are shown in Figure 56.

Figure 56. The positions of the buttons and text.



After the user clicks a button, the MessageDialog returns a MessageDialogButton object, which is either an ActionButton, CancelButton, or AlternateActionButton. By determining the type of object that was returned, you learn which button the user pressed. You can also examine the returned object's properties, if necessary.

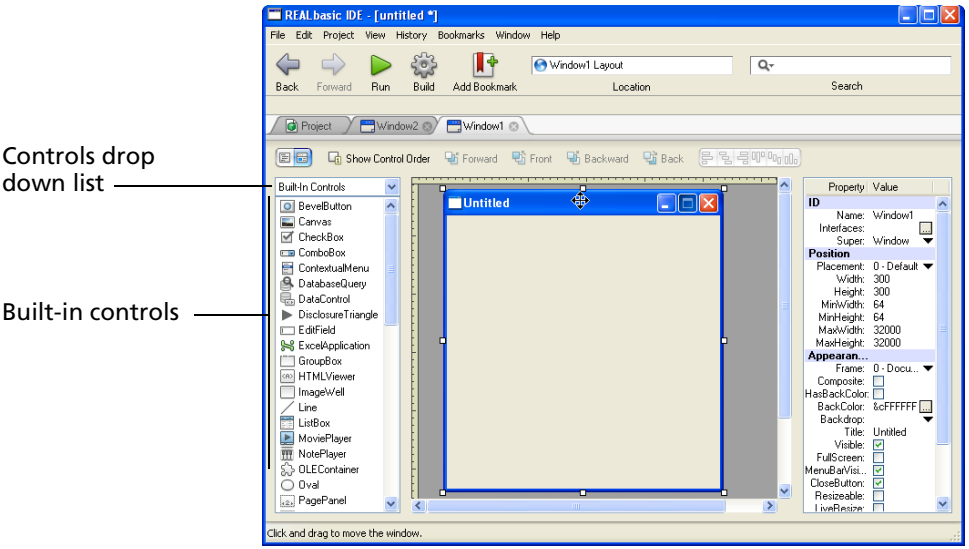
Figure 57 shows a sample MessageDialog box. See the entries in the Language Reference for the MessageDialog class and the MessageDialogButton classes for more information and examples.

Figure 57. The customized MessageDialog alert.

Interacting with the User Through Controls

Users provide information to your application through user interface controls. REALbasic provides a tremendous amount of flexibility in this area. Not only are there many built-in controls, but you can even create your own controls (you will learn more about this later). REALbasic's built-in controls are added to windows using the Controls pane, shown in Figure 58.

Figure 58. The Controls pane showing the Built-in Controls list.



The Controls drop-down list lets you select among three types of controls for display in the Controls pane, plus a customizable list of controls. The customizable list can be a mixture of controls from the three standard types. The three types of controls are Built-in controls, Project controls, and Plug-in controls. You use the Controls drop-down list to choose the type of controls that is displayed:

- **Built-in Controls:** The controls that are built into REALbasic. This is the default choice. The built-in controls are shown in Figure 58.
- **Project Controls:** Custom controls that are based on built-in controls. You can create customized versions of any of the built-in controls. If you do so, they appear in this list automatically. Project controls are also listed in the Project Editor. By default, this list has one item, Object. It is a "placeholder" item that you can convert to a control or object of any type. You do this by dragging an instance to the window and changing its Super property to the desired type of object. For information on creating custom controls, see the section "Understanding Subclasses" on page 421 and the section "Creating Custom Interface Controls with Classes" on page 455.

- **Plug-in Controls:** Controls that you added to REALbasic as by installing plug-ins. Third-parties can create custom controls in the form of plug-ins that are installed by placing the plug-in in the Plugins folder in the REALbasic IDE folder. This list is empty if you have no control plug-ins installed.
- **Favorites Controls:** Controls that you have added to your list of favorite controls. See the following section for information on creating your Favorite Controls list.

A few of the items in the Built-in Controls pane add objects to a window that are not visible to the end user. You can use them to add capabilities to the application that become available when the window is open. For example, you can use the TCPSocket control to support network communications via the TCP/IP protocol. Other controls support Microsoft Office Automation. They enable you to automate PowerPoint, Excel, and Word via REALbasic, but don't add a visible interface item to your application's windows. When you add one of these controls to a window, you then add code to the control.

Favorite Controls

To make the Controls pane more manageable, you can create a list of frequently used controls. Your Favorites list can include controls from all three built-in lists, the built-in controls, project controls, and plug-in controls. A shorter list that is made up of only the controls you use frequently will be easier to work with.

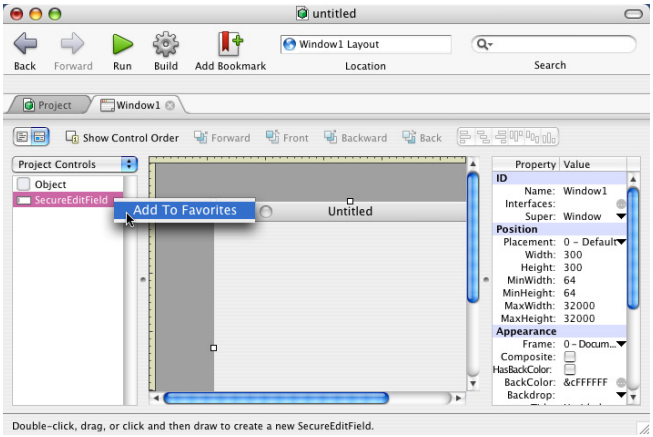
If you use project and plug-in controls as well as the built-in controls, you can combine all types of controls in your Favorites list of controls. For example, in Chapter 9 you will learn how to create a custom EditField controls that is based on the built-in EditField control but prohibits the user from copying the text in the control. When the customized control is first created, it appears only in the Project Editor and in the Project controls list, but you can use the Favorites list to add it to the same list as your favorite built-in controls.

To add a control to your Favorites list, do this:

1. **If it is not already displayed, use the Controls drop-down list to switch to the type of Controls list that contains the control you want to make a Favorite.**
2. **Right+click (Control-click on Macintosh) on the desired control.**

   A contextual menu appears.

Figure 59. The Add to Favorites contextual menu item.



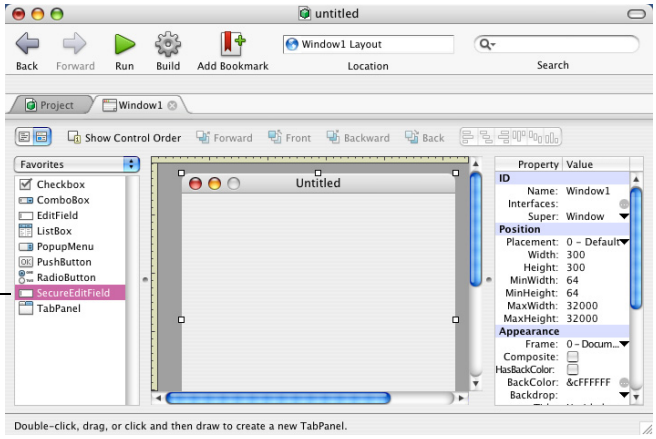
3 Choose Add to Favorites from the contextual menu.

When you switch to your Favorites list, the selected control will have been added to the end of the list.

4 Repeat this process to add additional controls from the current list or switch to another type of list and add additional controls.

For example, Figure 60 shows both Built-in and Project controls in a Favorites list.

Figure 60. A Favorites list with both Built-in and Project controls.



Adding, Changing, and Removing Controls

REALbasic makes adding, changing, and removing controls easy.

Adding Controls to a Window

There are several ways to add a new control to a window:

- Drag a control from the Controls pane to the Window Editor,
- Double-click a control in the Controls pane,
- Select a control in the Controls pane and draw a region in the Window Editor,

To add a control by dragging, do this:

1. **If the Window Editor for the window is not visible, click its tab or double-click its name in the Project Editor.**
2. **Display the desired type of Controls pane with the Controls drop-down list and then drag the control from the Controls pane to the desired location in the window and drop it onto the window.**

   REALbasic adds an instance of the control at the location of the drop.

To add a control by double-clicking, do this:

1. **If the Window Editor for the window is not visible, click its tab or double-click its name in the Project Editor.**
2. **Display the desired type of Controls pane with the Controls drop-down list and double-click the control.**

   REALbasic places an instance of the control in the window.

To add a control by selecting, do this:

1. **Click once on the control in the Controls pan to select it.**
2. **Press Enter (Return on Macintosh).**

   REALbasic adds an instance of the control at the default location in the window.

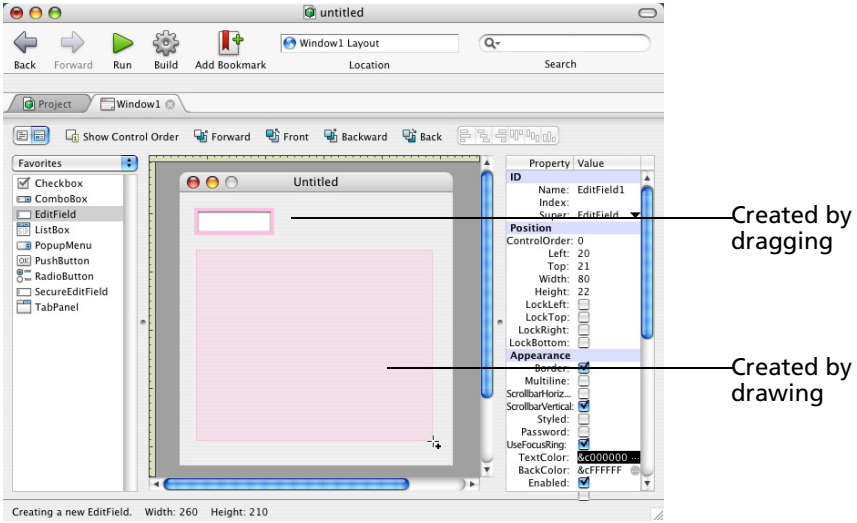
To add a control by drawing, do this:

1. **If the Window Editor for the window is not visible, click its tab or double-click its name in the Project Editor.**
2. **Display the desired type of Controls pane with the Controls drop-down list.**
3. **Click once on the control to select it and then use the pointer to draw a region in the window.**

   REALbasic places an instance of the control at the specified location and in the specified size.

   The last way of adding a control is especially convenient when the default size of the control is not appropriate. Suppose you want to add an EditField control to a window that will serve as a text editing area. If you use either of the first two methods of adding the control, REALbasic will use the default size for the EditField, which only permits one line of text entry. You will need to resize the control in another step. However, if you draw the region, you can specify both the location and size of the control in one step. Here is a comparison:

Figure 61. EditFields created by dragging and by drawing an area.



To use the contextual menu, do this:

1 **Right+click the mouse button in the window (Control-click on Macintosh).**

A contextual menu appears. The Add item is at the top level of a hierarchical menu of objects that can be added to the project. It displays the full REALbasic object hierarchy in the form of a hierarchical menu.

2 Use the Add menu item to choose the control you wish to add from the list of built-in controls.

The Add submenu item enables you to add any object to a window. The submenu command exposes the entire list of REALbasic classes and the REALbasic object hierarchy, which is indicated by multiple levels of submenus. When you choose an object that is not a control, the object is represented in the window by a generic REALbasic icon. When you click on the generic icon, it is selected in the window's Code Editor, just as if it were a visible control. Of course, it will not be visible in the built application.

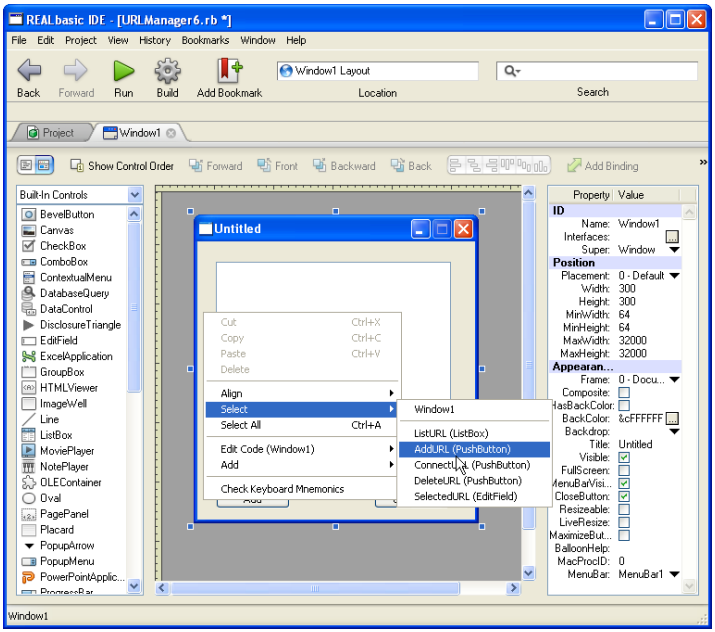
You can modify its properties in the Properties pane and add code to the object using the Code Editor, providing it has built-in events.

Selecting Controls

Controls can be selected in one of three ways: using the mouse button, the Tab key, or the contextual menu. The easiest way to select an individual control is to click on it. If you click on a control, it will be selected.

The contextual menu contains two items for selecting controls, Select and Select All. The Select All command selects all controls in the window and Select has a submenu that lists all of the window's controls. This enables you to select any control in the window. Choose a control from the submenu to select it.

Figure 62. Selecting a control with the window's contextual menu.



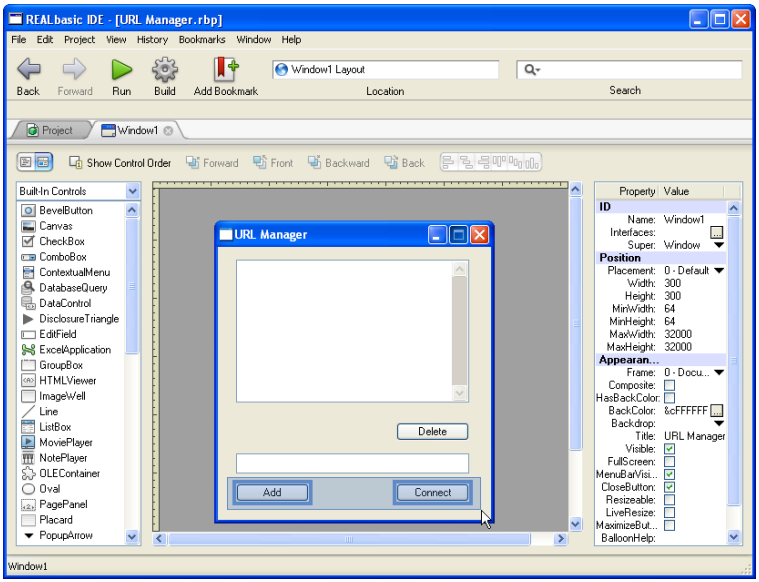
You can also move through the controls in a window by pressing the Tab key. Each time you press the Tab key, REALbasic will move from one control to another. This is also the order the user will move through the controls when using the Tab key. For more information, see "Changing The Tab (Control) Order" on page 146. Holding down the Shift key while pressing the Tab key selects controls in reverse Tab order. If only one control is selected, REALbasic draws resize handles at each corner of the control. You can select several controls by holding down the Shift key as you click on the controls. You can also draw a marquee around a group of controls to select them.

NOTE: You can also choose Edit ▶ Select All to select all the controls in the window. After all the controls are selected, you can Ctrl+click (Command-click on Macintosh) on a control to remove it from the selection.

You can select two or more controls by clicking on one control and then holding down the Ctrl key (Command key on Macintosh) and continue clicking on other controls. All the controls you click on are then selected.

If the controls you wish to select are next to one another, you can draw a selection rectangle around them using the mouse. Drag diagonally (i.e., from top-left to bottom-right) and release the mouse button. A translucent marquee surrounds the selected controls.

Figure 63. Selecting two controls using a selection rectangle.

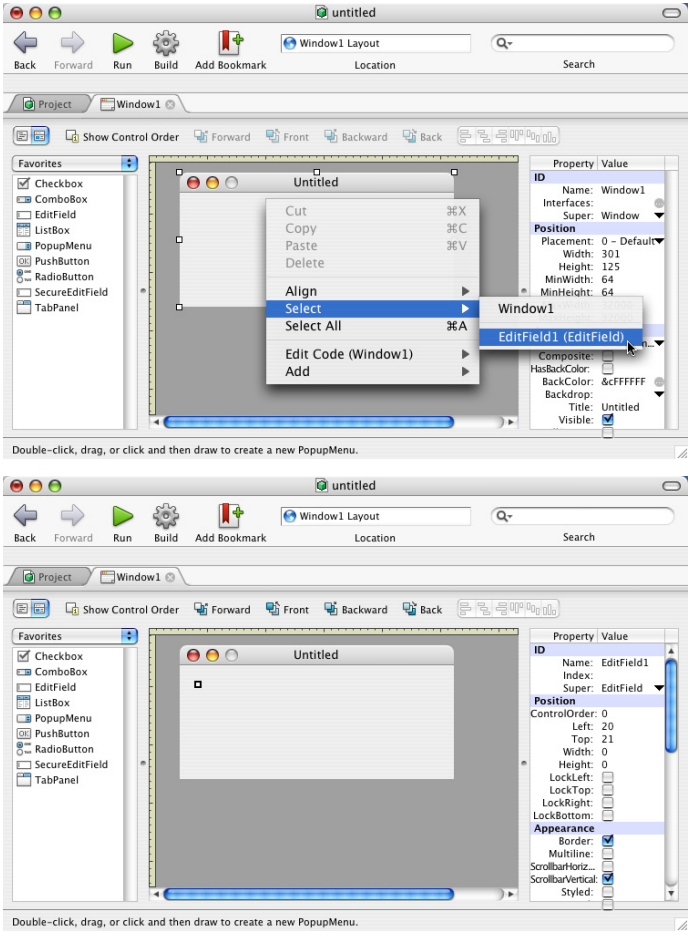


Selecting Invisible Controls

It's possible for a control to disappear. For example, if you give a control a large enough negative or positive Left or Top property, it will disappear off the edge of the window. Or, if you give it Width and Height properties of zero, it will remain in its position but become invisible. You would have a tough time clicking on it to select it.

However, you can always use the window's contextual menu to select an invisible control. The Select submenu will always list all controls that belong to the window even if they are not visible. In Figure 64, an EditField that has its Width and Height properties set to zero is selected in this manner. When selected, only a selection handle appears. Once it is selected, the Properties pane changes to show its properties. You can then use the Properties pane to change its position and size properties (see Figure 64. below and the following section).

Figure 64. Selecting an EditField that has zero width and height.



In the second illustration in Figure 64, notice that the Properties pane has changed to show the invisible EditField's properties. Its Width and Height properties are both zero. You can now change its Width and Height properties so that it will become visible.

Changing a Control's Position

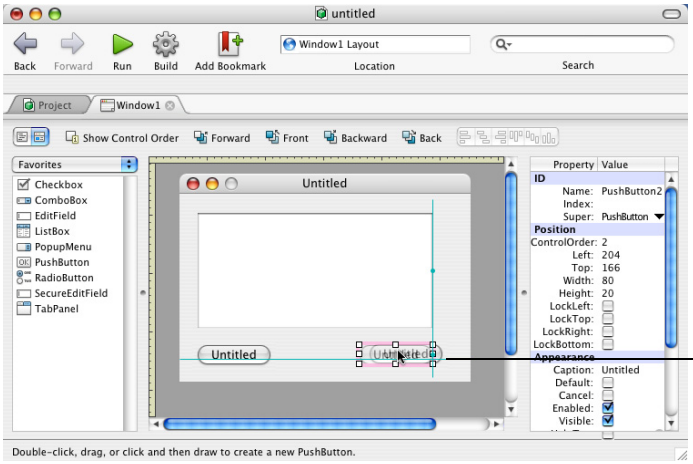
A control's position can be changed by dragging the control using the mouse, by using the arrow keys (to move it one pixel at a time in the horizontal or vertical directions) and by changing the properties in the Position group in the Properties pane.

The Left and Top properties determine the location of the top-left corner of the control, while the Width and Height properties determine its size. You can always use these properties to position and size controls precisely. For example, the invisible EditField in Figure 64 can be repositioned by changing its Left and Top properties.

Using Alignment Guides

When you drag a control, you can align it with other objects in the window by taking advantage of built-in horizontal and vertical alignment guides. When the object you are dragging is near the horizontal and/or vertical side of another object, alignment guides temporarily appear, allowing you to position the object precisely. Figure 65 on page 92 illustrates the process of aligning a PushButton with the baseline of another PushButton and the right side of an EditField control.

Figure 65. Alignment guides help you position objects.



Using the Lock "x" Properties to Set a Control's Position

Any visible control has four Boolean properties that you can use to "lock" the control's horizontal or vertical edges to the corresponding horizontal or vertical edges of the window. These properties are LockLeft, LockRight, LockTop, and LockBottom. When one of these properties is turned on, the space between the designated edge of the control and the corresponding edge of the window remains the same when the user resizes the window.

You use these properties to tell REALbasic to resize or move the control when the user resizes the parent window. For example, if you use a multiline EditField as a text processing window, you will want to align the edges of the control to the window and then use the LockLeft, LockRight, LockTop, and LockBottom properties to resize the control automatically when the user resizes the window. Figure 66 on page 93 illustrates this:

Figure 66. A multiline EditField used as a text processor.

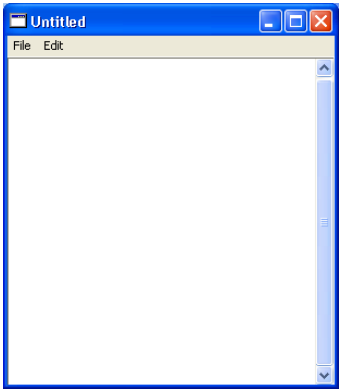
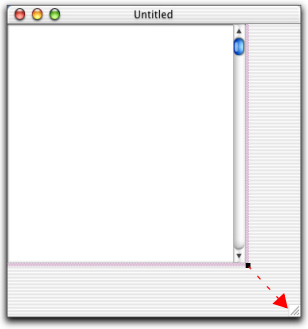
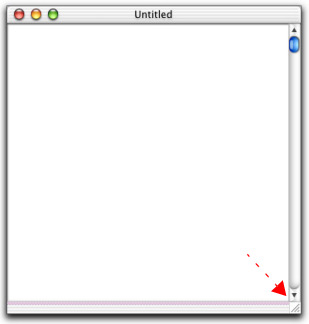


Figure 67 on page 93 shows how these four properties work.

Figure 67. Resizing the window with the "Lock x" properties on and off.



Lock properties off: the EditField maintains its size and position when the window is resized.

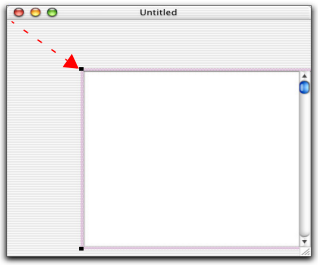


Lock properties on: the EditField grows and shrinks as the window is resized.

In this example, if only the LockRight and LockBottom properties are set, the EditField will move down and to the right as the window's grow box is moved in that direction.

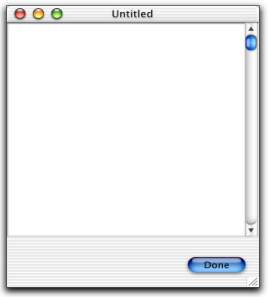
Figure 68. Enlarging a window when LockLeft and LockTop are not set.

Only the EditField's bottom and right sides are locked.



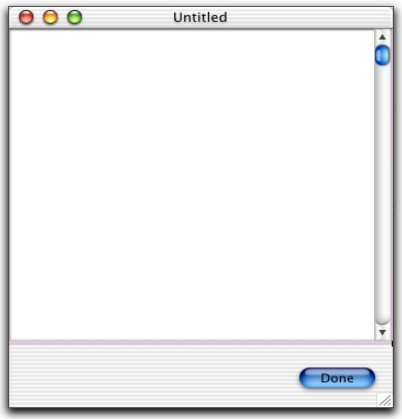
The situation gets more interesting as we add another control. Consider the layout in Figure 69.

Figure 69. A PushButton below the EditField.

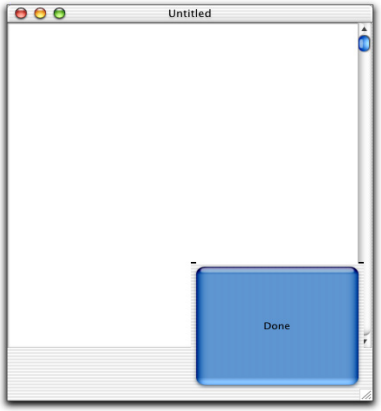


The PushButton should be locked to the bottom and right side of the window only; we want it to maintain its relative position to the EditField. However, we do not want to resize the PushButton as the window is resized. This is illustrated in Figure 70 on page 94.

Figure 70. Effects of resizing a window when a PushButton is locked.



The PushButton is locked to the bottom and right side; it maintains its shape and relative position to the EditField as the window is enlarged.



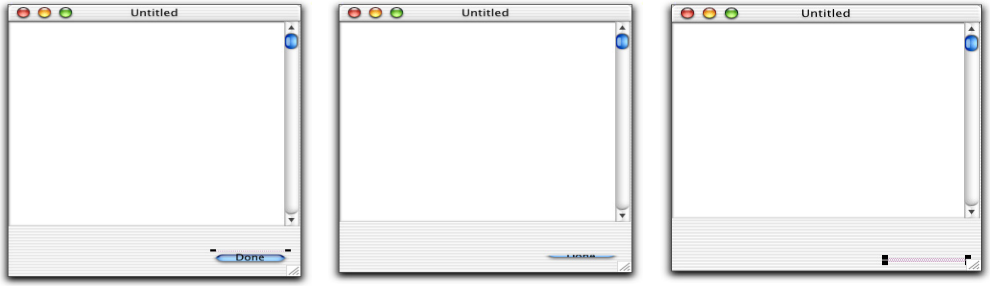
The PushButton is locked to all four sides; it maintains the vertical and horizontal distances to the left and top of the window, forcing it to grow as the window is enlarged.

An unusual effect is achieved if the user reduces either the horizontal or vertical dimension of the window when the PushButton is locked to the sides that are getting closer to each other. In Figure 71 on page 95 the vertical dimension is decreasing and the control is locked to both the top and bottom.

Since the PushButton is supposed to maintain its distance from both the top and bottom, the control can appear to be pushed right out of existence as the height of the window is reduced. In Figure 71 on page 95, the PushButton is trying to

maintain its distance from both the top and bottom edges of the window simultaneously as the height of the window decreases. Eventually, it will collapse into REALbasic's version of a Black Hole. In the right image in Figure 71, the PushButton has folded up upon itself (i.e. in effect, its bottom edge is above its top edge).

Figure 71. Effect of reducing the height of a window on a PushButton that is locked to both the top and bottom.



Going... going... gone.

This happens because there is space both above and below the PushButton. The EditField, on the other hand, doesn't get pushed out of existence because there is no space between the top of the control and the top of the window in Figure 69 on page 94.

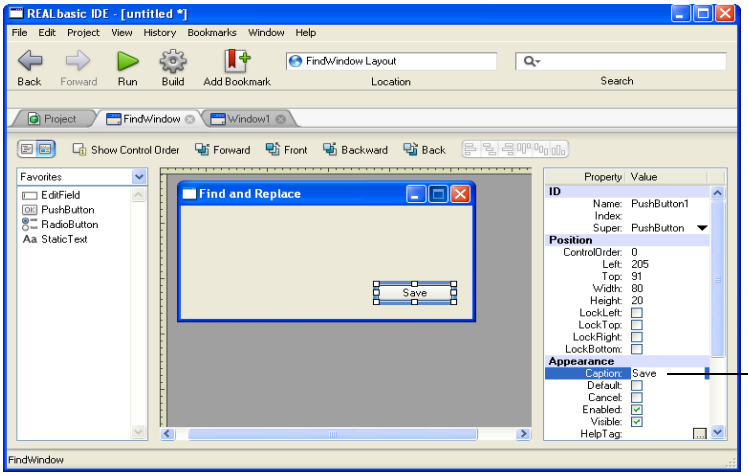
Changing a Control's Properties with the Properties Pane

Some changes to a control must be made without the Properties pane. For example, controls can be rearranged by simply dragging them from one place to another inside the window. However, most of the changes you make to controls will be made using the Properties pane.

The Properties pane displays the properties of the currently selected control *that can be changed from the Development environment*. If more than one control is selected, the Properties pane displays only those properties common to all of the selected controls.

Some properties are entered by typing, while others with on/off-type values are represented by a CheckBox. If the property is set by typing, you can use either the Enter or the Return key to commit the new value. Some controls' Caption or Text properties can be set selecting the control in the Window Editor and typing the new text immediately.

Figure 72. Changing the Caption property of a PushButton.



Numeric Properties

If the property you want to set is numeric — such as the Left, Top, Width, or Height properties or the number of columns in a ListBox — you can either enter a number or an expression that evaluates to a number.

If you want to enter an expression, your available operators are: +, -, *, /, \ (integer division), % (mod), and ^ (power). You can also use parentheses to control the order in which subexpressions are evaluated. You can also use references to property values by name, allowing you to write expressions such as "Top*2" or "Width+Left."

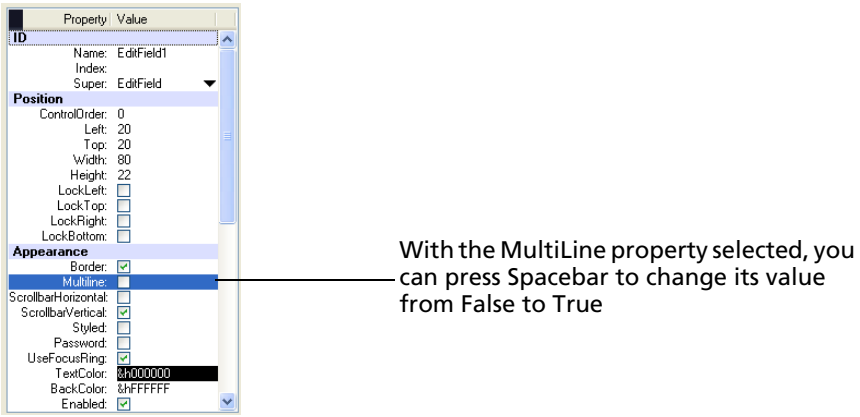
If an expression is invalid on its own, the current value of that property will be prepended; this allows you to (for example) enter "*2" as a handy shortcut for doubling the current value when multiple objects are selected. To add a value to the current value, use "+ +*value*". For example, to add 10 use "+ + 10", since "+10" will be treated as "10".

Boolean Properties

The values of Boolean properties are shown as CheckBoxes in the Properties pane. A value of False is indicated by an unchecked CheckBox and a value of True is indicated by a checked CheckBox.

You change a value from False to True or vice versa by clicking on the CheckBox. Or, you can select the boolean property by clicking on its name and then pressing the Spacebar. In Figure 73 the user has selected the MultiLine property by clicking on its name. Pressing SpaceBar will toggle this property's value.

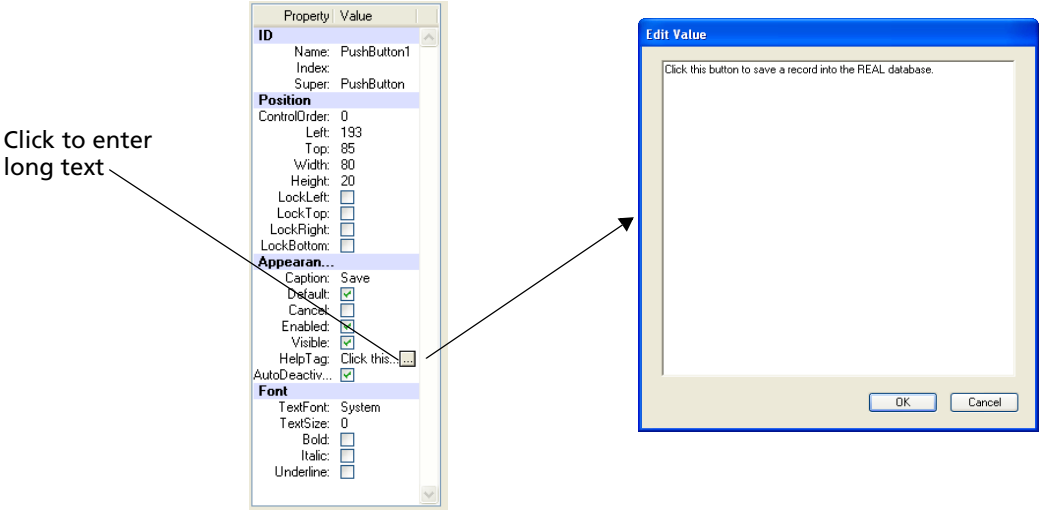
Figure 73. Selecting a Boolean property.



Text Properties

Properties such as the text that appears in an EditField or the caption of a PushButton are entered simply by typing into the text area. If the text you want to enter is long, you can click the text icon to bring up a modal window into which you can enter more lengthy text. A typical Text window is shown in Figure 74.

Figure 74. Entering text for HelpTag in a separate window.



Constants

You can also choose to enter a constant as a value in the Properties pane. For example, you can create a constant that contains the caption for all of the "Accept" PushButtons in your application. If you want to change the caption, you only need to change the value of the constant rather than edit the values for each PushButton.

Constants are especially useful for applications that are deployed in more than one language. Instead of using literal text as property values, you use constants for all

text that the user sees. This includes menus and menu items as well as windows. REALbasic enables you to use different values of each constant for different languages.

A good place to create constants for this purpose is in a module. A constant can be given Global scope only in modules. In Figure 75 a global constant is being created in a module and then referred to in the Caption property of a PushButton.

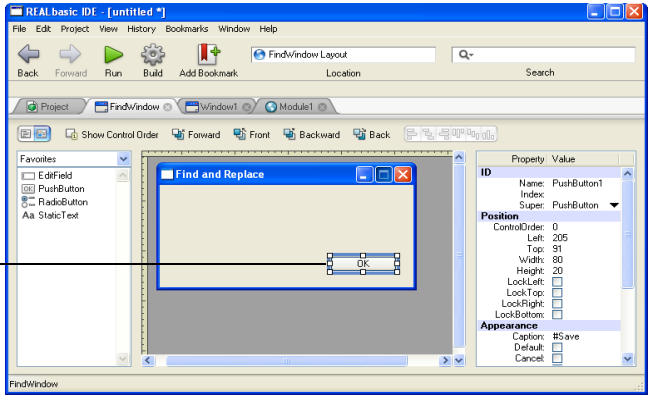
To use a constant, precede the name of the constant by the number sign, #, as the value for a property in the Properties pane. For example, if you want to use a global constant (in a module) named "Save", you would refer to it in the Properties pane as "#Save". If it were a public constant in Module1, you would refer to in the Properties pane as "#Module1.Save'.

Here are the Appearance properties of a PushButton that references the Global constant named "Save."

Figure 75. Setting the Caption property of a PushButton using a constant.

The constant "Save" is defined as the string "OK" in a module...

...and is used as the Caption property of a PushButton



The control in the Window Editor shows the value of the constant as defined in the module

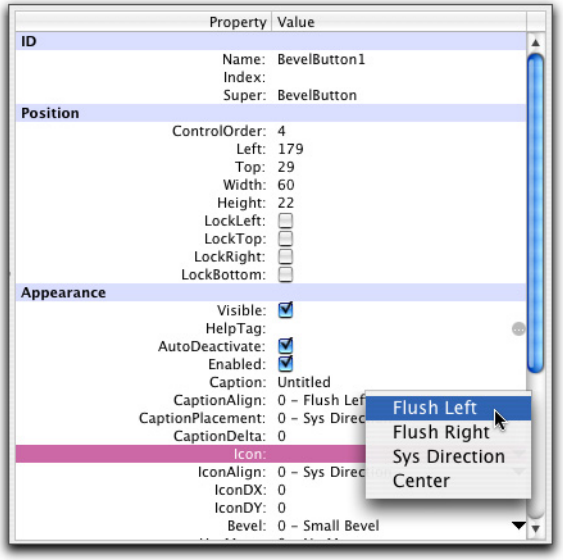
In the built application, the PushButton uses the value of the "Save" constant.

For information about creating constants and using them to localize your application, see the section "Using Constants to Localize your Application" on page 307.

Choice Lists

Some properties that require you to choose a value from a fixed list are displayed as pop-up menus. Such properties have a downward-pointing arrow to the right. Simply choose the desired value from the pop-up menu. In Figure 76, the "Right of Graphic" value is being assigned to the CaptionPlacement property of a BevelButton control.

Figure 76. Choosing a value from a Property pop-up menu.

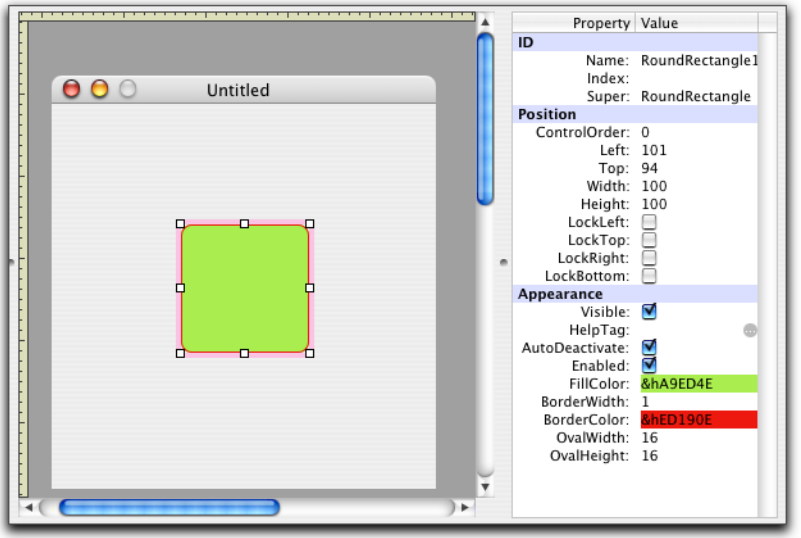


Color Properties

Color properties display the selected color. These colors can be changed by clicking on the color in the Properties pane and using the Color Picker to choose a color. If you wish, you can enter the color value by supplying RGB (Red-Green-Blue) values using the & operator.

For example, the Rectangle and Rounded Rectangle controls have properties that control the border and fill colors. You can set these colors by clicking on the color swatch in the Properties pane to choose the color using the Color Picker. In Figure 77 the RGB value of the color is also shown in the FillColor and BorderColor properties. For information on setting the color by specifying the RGB color model, see the section "Color" on page 173.

Figure 77. The Properties pane for a Rounded Rectangle.



Removing Controls

You can remove a control from a window using the control's contextual menu or a menu command.

To remove a control from a window, do this:

1. **Bring the window that contains the control to the front. If it's not open, double-click on it in the Project Editor to open it.**
2. **Click on the control to select it.**
3. **Choose Edit ▶ Cut (Ctrl+X or ⌘-X on Macintosh), or Edit ▶ Delete or press the Delete key, or right+click on the control (Control-click on Macintosh) and choose Cut or Delete from the contextual menu.**

   If you want a copy of the deleted control on the Clipboard, use Cut instead of Delete.

Understanding Control Layers

Each control in a window has its own layer. This layer is like a sheet of transparent plastic on which each control is placed. It determines whether one control is in front of the other. The Window Editor toolbar provides commands for moving a control forward one layer, to the front, backwards one layer, and to the very back of the layers. These commands are also available in the Edit▶ Arrange submenu.

Control layers determine the order in which your application selects the controls as the user presses the Tab key. For more information, see the section "Changing The Tab (Control) Order" on page 146 for more details. Control layers may also be important when controls overlap.

Understanding The Focus

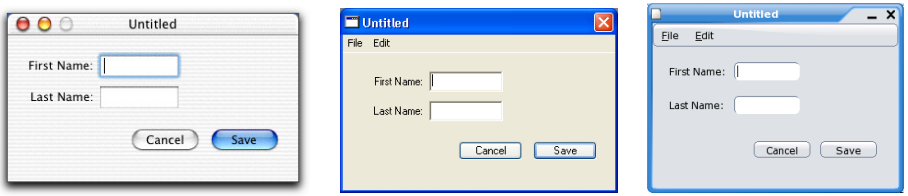
The *focus* is a visual cue that tells the user which control receives keystrokes. EditFields, ComboBoxes, Canvas controls, PushButtons, and ListBoxes can receive the focus on Macintosh. In addition, Sliders, PopupMenus, and CheckBoxes can also receive the focus on Windows. On Linux, EditFields, ComboBoxes, CheckBoxes, PushButtons, PopupMenus, and Sliders can receive the focus.

EditFields

EditFields on any platform display the focus by showing a blinking insertion point and accepting text entry. The behavior of the EditField when the Tab key is pressed is controlled by the AcceptTabs property. If this property is False, pressing the Tab causes the EditField to lose the focus and the next control in the entry order gains the focus. If AcceptTabs is True, the EditField accepts the Tab character for data entry, just as any text character. The EditField keeps the focus.

In Figure 78, the First Name EditField has the focus. On Windows and Linux, the EditField does not get a focus ring; the focus is indicated only by the blinking insertion point.

Figure 78. EditFields with and without the Focus.



ComboBoxes

ComboBoxes on any platform display the focus by highlighting the selected item and showing a blinking insertion point. On Macintosh, a ComboBox with the focus also has a focus ring. In other words, a ComboBox with the focus acts like an EditField that has a pop-up menu attached to it.

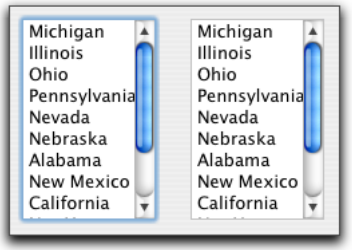
Figure 79. ComboBoxes with and without the focus.

When a ComboBox has the focus, you can scroll through the list of choices with the up and down arrow keys and select an item by pressing the Return key. Of course, you can also edit or replace the selected item.

ListBoxes

When a ListBox has focus on Macintosh, REALbasic draws a focus ring around the ListBox. If the user is running Mac OS "classic," the focus ring is drawn in the operating system's Accent Color.

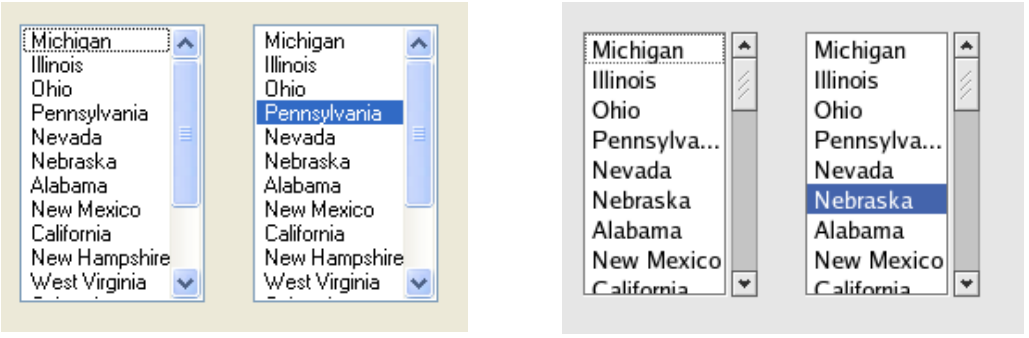
Figure 80. A ListBox on Macintosh.



With the Focus | Without the Focus

When a ListBox gets the focus on Windows or Linux, REALbasic either draws a marquee around the previously selected item (i.e., the highlighted item) or the first item in the ListBox if there is no previously selected item. The marquee may be difficult to see because the item is also highlighted. If the previously selected item is not currently displayed (i.e., the user has scrolled it out of view), there is no visible change in the appearance of the ListBox.

Figure 81. A ListBox with the focus (Windows and Linux).



Windows, no previous selection | Windows, previous selection | Linux, no previous selection | Linux, previous selection

When a ListBox has the focus, it responds to the Up and Down arrow keys. Pressing either arrow key changes the selected (highlighted) text. It also receives any other keys the user types. This allows you to provide *type selection* functionality where typing selects the item that matches the characters being typed. An example of type selection is provided with REALbasic.

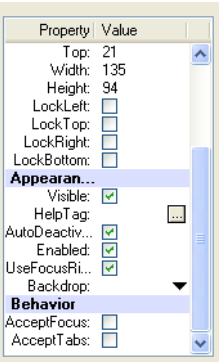
NOTE: If a ListBox is the only control in the window that can get the focus, it will initially get the focus and keep it.

Canvas Controls

Since the Canvas control can get the focus, you can use it to create custom controls of any type that can get the focus. For example, you can use the Canvas to simulate controls that don't get the focus on Macintosh, such as buttons and pop-up menus. Unlike other controls that can get the focus, the ability of a Canvas to accept the

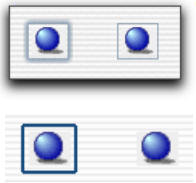
focus is turned off by default. In order for a Canvas control to receive the focus, you must set the AcceptFocus property to True. This can be done either in code or in the Properties pane. The Canvas control also has an AcceptTabs property that indicates whether pressing Tab selects the next control in the window or sends the Tab keystroke to the Canvas control for processing. If AcceptTabs is off, pressing Tab causes the Canvas control to lose the focus. The next control in the entry order gets the focus. If AcceptTabs is on, the Canvas control detects the Tab key as if it were any other key and allows your code to detect and respond to the Tab key.

Figure 82. The Canvas Control's properties relating to focus.



If the AcceptFocus and UseFocusRing properties are set to True, the Canvas control indicates focus on Macintosh by drawing a border around the control. If the user is running Mac OS 8 "Classic," the border is drawn in the operating system's Accent Color. This is shown in Figure 83.

Figure 83. Canvas controls with and without the focus.



On Windows and Linux, unfortunately, the UseFocusRing property has no effect. There is no visual indicator of focus that works automatically. However, it is easy to simulate a focus ring using the language. See the Canvas control entry in the *Language Reference* for an example of how to do this.

PopupMenus

On Windows, when a PopupMenu receives the focus the currently selected item is highlighted. Like the ListBox, it also responds to the up and down arrow keys and provides the same type selection functionality. Figure 84 on page 104 shows a PopupMenu loaded with the list of states shown in the ListBoxes in Figure 82 on page 103 with and without the focus. If the user types an "O" while the

PopupMenu has the focus, the state of Ohio is selected; typing a "C" selects California, and so forth.

Figure 84. A PopupMenu with and without the focus on Windows.

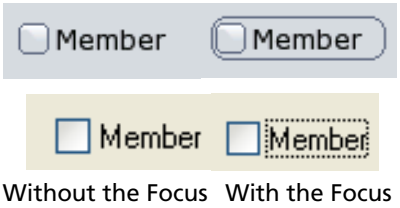
When a PopupMenu gets the focus on Linux, an insertion point appears in the current menu item. You can change the selected menu item with the up and down arrow keys.

Figure 85. A PopupMenu with and without the focus on Linux.

CheckBoxes

When a CheckBox gets the focus, a marquee surrounds the CheckBox label. Pressing the Spacebar while the CheckBox has the focus toggles the control between its unchecked and checked states.

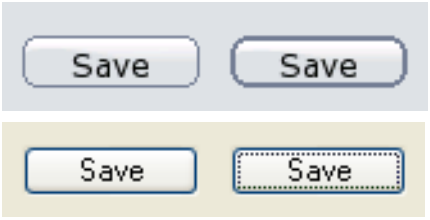
Figure 86. A CheckBox with and without the focus on Linux and Windows.



PushButtons

When a PushButton gets the focus, a marquee surrounds the PushButton's caption. Pressing the Spacebar while the PushButton has the focus pushes the button, i.e., executes its Action event handler.

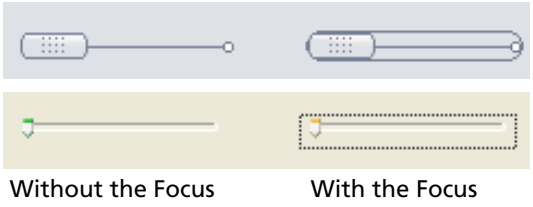
Figure 87. A PushButton with and without the focus on Linux and Windows.



Sliders and Scrollbars

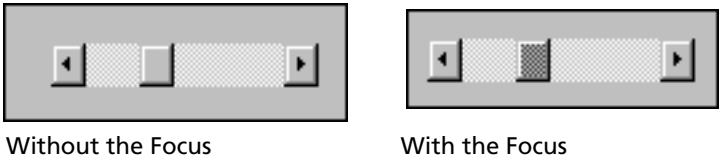
When a Slider gets the focus, a marquee surrounds the control. Pressing the Up or Left arrow key decreases the value of the Slider and pressing the Down or Right arrow key increases the value of the Slider. The amount that the value is changed by each keypress is controlled by the Slider's LineStep property. By default, the Slider has a range of 0 to 100 and LineStep is 1. The user can also click anywhere along the slider's track to change the slider's value. The amount that the slider moves with each click is controlled by the Slider's PageStep property. The default value of PageStep is 20.

Figure 88. A Slider control with and without the focus.



When a ScrollBar gets the focus on Windows, the thumb begins to pulse. This is shown in Figure 89.

Figure 89. A ScrollBar with and without the focus.



The ability of a ScrollBar to get the focus can be turned off by deselecting its AcceptFocus property.

BevelButtons

When a BevelButton gets the focus, a selection rectangle surrounds its label. When it has the focus, the user can press the button by pressing either the Spacebar or the Enter key. BevelButtons can get the focus only on Windows. You must set the AcceptFocus property to True to enable a BevelButton to get the focus.

Figure 90. A BevelButton with (left) and without the focus.

Disclosure Triangle

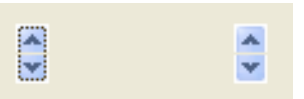
When a DisclosureTriangle gets the focus, a selection rectangle appears around the control. The user can toggle its state by pressing either the Spacebar or the Enter key. You must set the AcceptFocus property to True to enable a DisclosureTriangle to get the focus. DisclosureTriangles can get the focus only on Windows.

Figure 91. A DisclosureTriangle with (left) and without the focus.

UpDownArrows

When an UpDownArrows control gets the focus, a selection rectangle appears around the control. The user can press the Up and Down arrow keys on the keyboard to press the top and bottom arrows in the control. You must set the AcceptFocus property to True to enable an UpDownArrows control to get the focus. UpDownArrows can get the focus only on Windows.

Figure 92. An UpDownArrows control with (left) and without the focus.

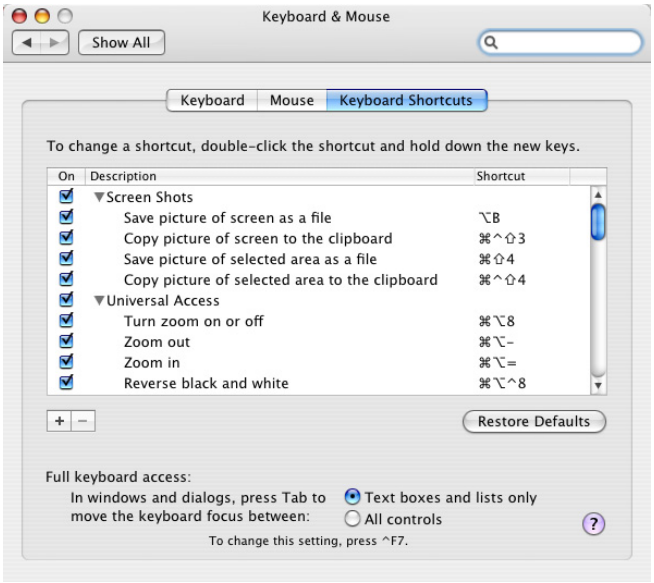


Full Keyboard Access

Mac OS X includes a system-wide feature called *full keyboard access*. This feature enables a user to do work with only the keyboard that ordinarily is done using both the keyboard and the mouse. For example, the standard Mac OS X interface calls for mouse gestures to operate the menu system and the dock. With full keyboard access, when the menu bar has the focus, menu items can be highlighted by the up and down arrow keys and an item is selected by pressing Spacebar.

Full Keyboard Access is enabled in the Keyboard Shortcuts panel of the Keyboard and Mouse System Preference. Click the All Controls radio button, shown in Figure 93, to enable Full Keyboard Access.

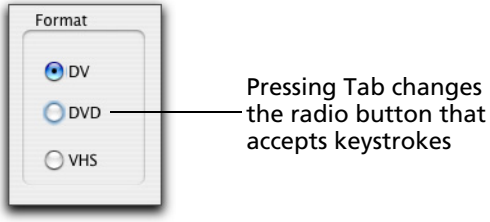
Figure 93. Enabling Full Keyboard Access (Mac OS 10.4).



When full keyboard access is on, you can select and set values for controls via the keyboard that normally do not have the focus. When a control accepts keystrokes via full keyboard access, it has a halo. For example, when full keyboard access is on, the user can select radio buttons via the keyboard only.

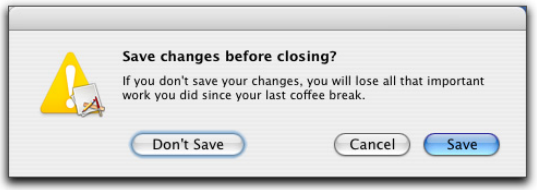
In the example shown in Figure 94, full keyboard access is on and radio buttons can get the focus. The user can press Tab to change the focus. In Figure 94 the "DVD" radio button accepts keystrokes. Pressing the Spacebar key sets the value of the radio button with the focus.

Figure 94. Selecting a radio button with full keyboard access.



When full keyboard access is on, PushButtons can be selected by pressing the Tab key and "clicked" by pressing the Spacebar. In Figure 95, the Don't Save button has the focus, denoting that it accepts keystrokes but the Save button is the default button. Pressing Spacebar with the dialog box in this state is equivalent to clicking the Don't Save button.

Figure 95. Selecting the Don't Save button via the keyboard.



To "click" a MessageDialog button using full keyboard access, press and hold down the Spacebar. Pressing the Spacebar once will highlight the button as if pressed but will not dismiss the dialog.

Please note that full keyboard access is a Mac OS X system-wide option that the end-user must select on his machine. The Mac OS X version of REALbasic supports full keyboard access if the user chooses to turn it on. However, you cannot turn it on for them, so you can't assume that all (or any) of your users will be using full keyboard access.

Duplicating Controls

You can duplicate the selected control or controls by choosing Edit ▶ Duplicate (Ctrl+D or ⌘-D on Macintosh) or by holding down the Option key and dragging the selected control (Macintosh).

The Object Hierarchy

Since controls are objects and REALbasic is an object-oriented language, all controls are derived from other classes. Among other things, this means that each control inherits properties from the class that it is derived from. Built-in controls that are visible are subclassed from the RectControl class, which means that each such control inherits a group of properties from the RectControl class automatically. (A few of the built-in controls are not visible in built applications, such as the Timer and TCPSocket.)

In the *Language Reference*, the parent class for a control is listed as the Super Class and has a hyperlink to the Super Class. To view all the properties and methods for a control, you need to check out the properties and methods of the Super Class as well as the properties and methods that are unique to the control. If a control's Super Class has another Super Class, then you need to keep going up the hierarchy.

For example, the RectControl class is the Super Class for visible controls. It has the properties Width, Height, Top, and Left which are used to establish the position and size of the RectControl in the window. Since all controls that are derived from RectControl have these properties automatically, they are not repeated for each type of RectControl in the Language Reference.

To see the object hierarchy in the REALbasic IDE, use a Window Editor's contextual menu and choose the Add menu item. The object hierarchy is shown as a hierarchical menu system.

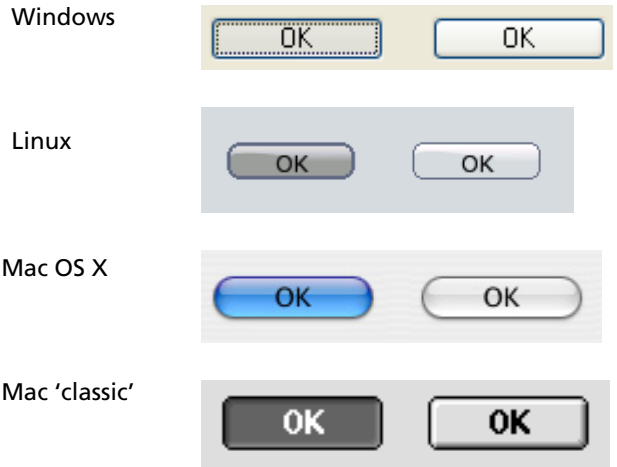
Button Controls for Performing Actions

There are four controls that are commonly used to perform actions when clicked: the CheckBox, the PushButton, the BevelButton, and the RadioButton. They are derived from the RectControl class.

PushButton

When clicked, a PushButton appears to depress giving the user feedback that they have clicked it. PushButtons are typically used to take an immediate and obvious action when pressed, like printing a report or closing a window. PushButtons can have the focus on Windows.

Figure 96. A PushButton pressed and unpressed.



BevelButton

The BevelButton control provides very similar functionality as the PushButton and adds several additional powerful features. You can, for example:

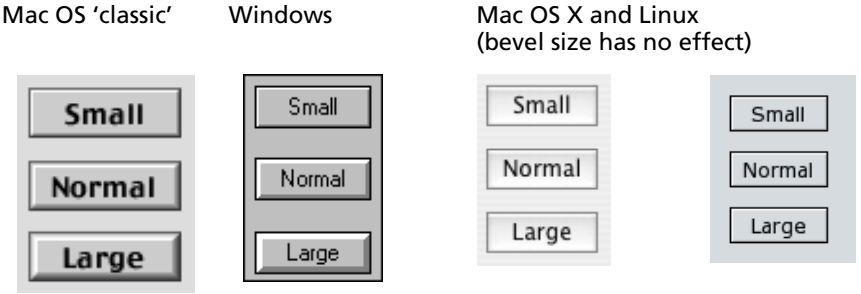
- Add an image to the control,
- Control the alignment of the button's text and/or the positioning of the text with respect to the graphic,
- Add a popup menu to the control,
- Control the feedback the user receives when the BevelButton is clicked.

The usage of a BevelButton control as a pop-up menu is described in the section "BevelButton" on page 121. Note that you can combine a PICT image with a pop-up menu.

Here are several examples of BevelButton options:

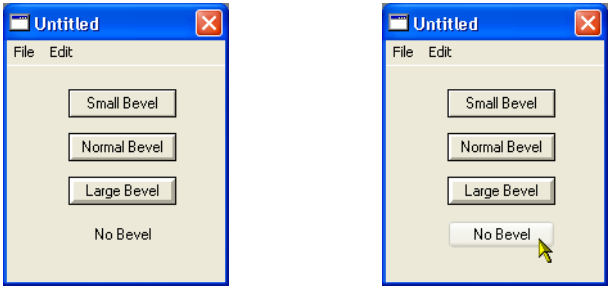
Figure 97. Icon, Text, and 'combo' BevelButtons.

Figure 98. Bevel Sizes.



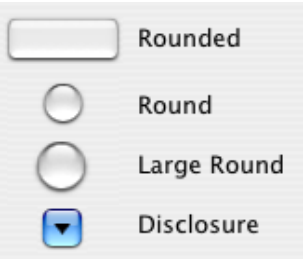
On Windows XP only, the 'No Bevel' option supports 'mouse-over' effects. When the mouse is not in the region of the button, only the button text is visible. When the mouse enters the region of the button, the button itself is shown. On all non-XP operating systems, the 'No Bevel' option appears the same as 'Small' in Figure 98. On XP, a BevelButton with the 'No Bevel' size selected behaves as shown in Figure 99.

Figure 99. A BevelButton on Windows XP with the 'No Bevel' style.



On Mac OS X only, four additional bevel styles are available: Rounded, Round, Large Round, and Disclosure. When clicked, the Disclosure bevel style toggles between two states: upward and downward pointing arrows. The others highlight when clicked. If deployed on other platforms, these bevel styles all look like a standard Small bevel BevelButton. The Mac OS X-only bevel styles are illustrated in Figure 100.

Figure 100. The Mac OS X-only BevelButton bevel styles.

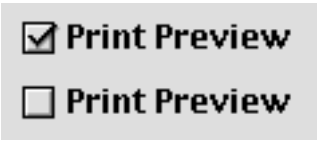


CheckBox

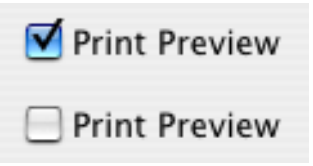
CheckBoxes are used to let the user state a preference that has only two possible choices, in which one of the choices can be selected by default. CheckBoxes should not cause an immediate and obvious action to occur except perhaps to enable or disable other controls.

Figure 101. A CheckBox checked and unchecked.

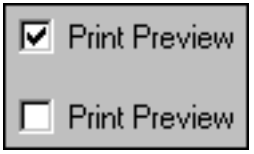
Mac 'classic'



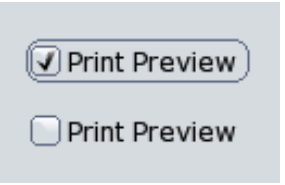
Mac OS X



Windows



Linux



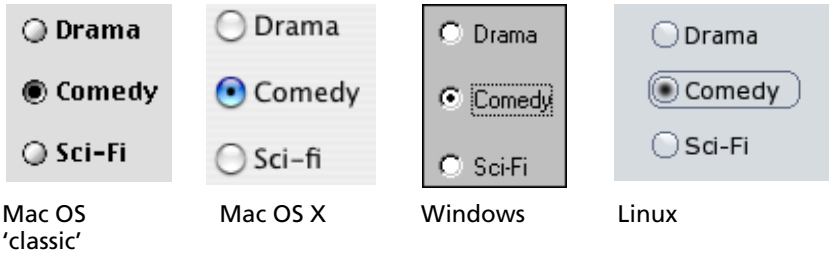
If space permits, consider using two RadioButton controls instead of a single CheckBox control as it will make the user's choice more obvious especially to the new computer user.

RadioButton

RadioButtons are used to present the user with two or more choices, where one of the choices can be selected by default. Selecting one RadioButton causes the RadioButton that is currently selected to become unselected. They are called RadioButtons

because they act just like the row of buttons for changing radio stations on car radios. Pushing one button deselects the current radio station and selects another station. RadioButtons should always be displayed in groups of at least two.

Figure 102. A group of RadioButtons with one selected.



Mac OS 'classic' | Mac OS X | Windows | Linux

If you are creating a window that will have two or more independent sets of RadioButtons, you will need to use a GroupBox control to make your RadioButton groups respond independently. See "GroupBox" on page 122.

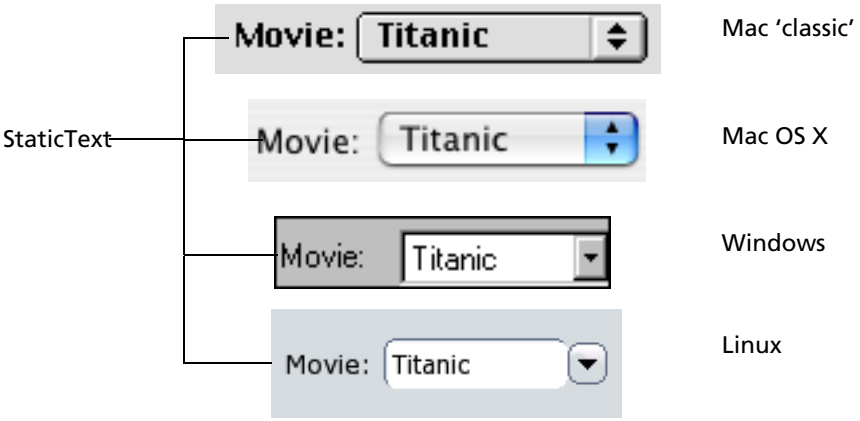
Controls for Displaying and Entering Text

REALbasic provides controls that let you display text the user can't select, display text the user can select but not edit, and display text the user can both select and edit. These controls are also derived from RectControl.

StaticText

Used to display text that the user cannot select or edit. StaticText controls are commonly used to label other controls (like PopupMenus) or provide titles for groups of controls. The text of the label can be controlled via code (it is the Text property of the control), so you can use it to display dynamic text that is "read only." For example, read-only fields from a REAL database can easily be displayed with a StaticText control. An easy way to do this is to use the StaticText control in conjunction with a DataControl. The DataControl enables you to "bind" the StaticText to a field in the database and display the contents of the field as the user navigates among records.

Figure 103. A StaticText control used to label a PopupMenu control



EditField

EditFields display text that can be modified by the user. It can be configured either as a "single line" EditField, such as a database field, or a multiline EditField, such as a word processor field. A multiline EditField can display text in multiple fonts, styles, and sizes and have both horizontal and vertical scrollbars. Individual paragraphs can be left, centered, or right aligned via the StyledText class. See the entry in the *Language Reference* for the StyledText class for more information on how to create the example shown in Figure 105 on page 114.

A single-line EditField can be configured a "password" field — displaying a bullet for each character that is entered. You can also specify a mask for the EditField which filters data entry on a character-by-character basis. For example, if you are using an EditField for a telephone number entry area, you can specify that only numbers can be entered and you can restrict the entry to the correct number of characters.

Figure 104. A Single-line EditFields used for read/write access to database fields.

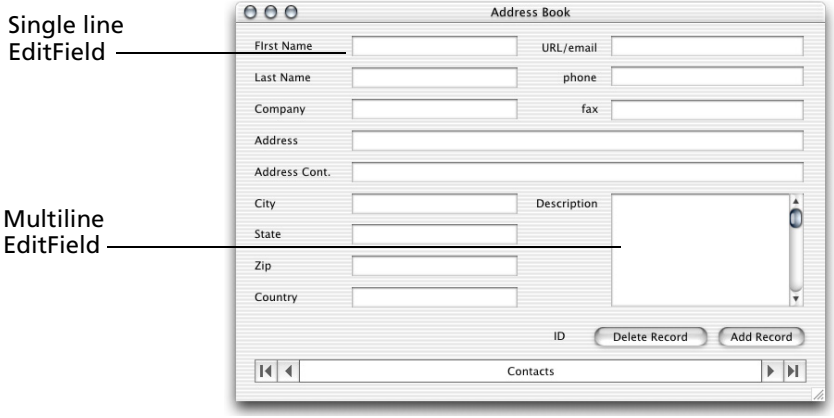
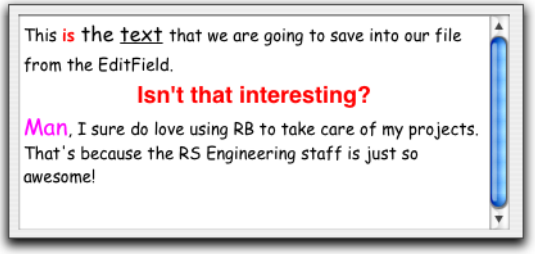


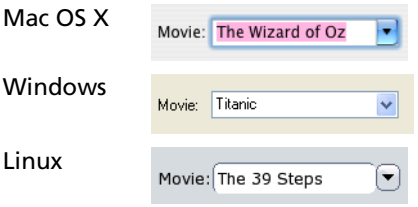
Figure 105. An EditField with multiple fonts, styles, sizes, colors and paragraph alignments.



ComboBox

The ComboBox control works like a combination of a single-line EditField and a pop-up menu. The user can either enter text in the ComboBox or choose an item from the attached pop-up menu. Also a menu selection can be selected and modified. Unlike a real EditField, you cannot use a mask to filter data entry or operate it as a Password field.

Figure 106. A ComboBox on Macintosh, Windows, and Linux.



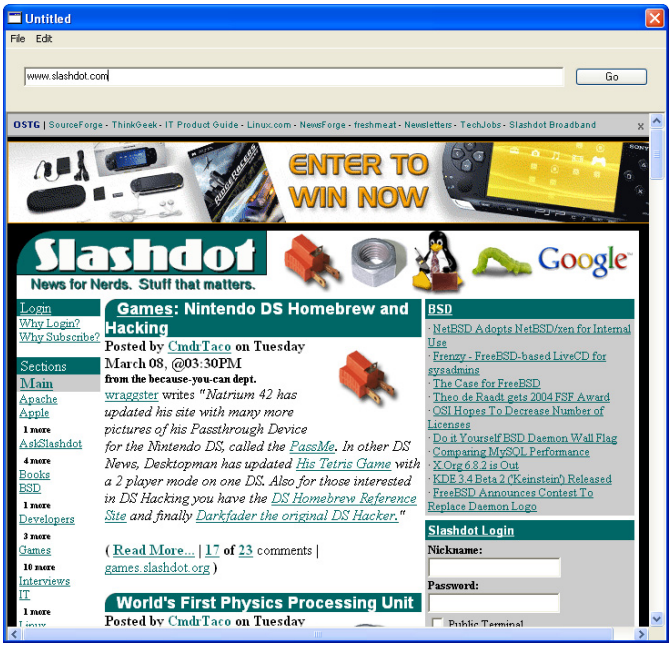
For some examples and information about using a ComboBox to present a list of choices, see the section “ComboBox” on page 121.

HTMLViewer

The HTMLViewer control renders HTML (like any web browser application) and provides basic navigation functions. Using the language, you can pass it an HTML file, the HTML text itself, or tell it to load the HMTL specified by a URL. If the HTML is valid, it renders it. The REALbasic Online Help uses the HTMLViewer control.

The following example is a very simple web browser that uses an HTMLViewer control. The user types in a URL into the EditField at the top of the window and clicks the Go button. If it is a valid URL, the web page appears in the HTMLViewer control.

Figure 107. A very simple web browser that uses an HTMLViewer control.



Controls for Displaying and Entering Numeric Values

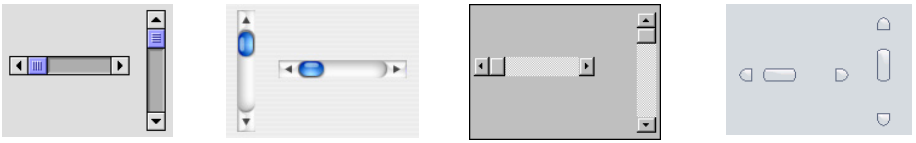
REALbasic provides controls that can be used to let the user choose a numeric value from a range or to display a numeric value from a range. In some cases, these controls can also be used to control the display of another control. For example, a ScrollBar control might be used to determine which portion of a picture in a Canvas control is displayed (in other words, act as the Canvas control's scrollbar).

ScrollBar

ScrollBars can be presented vertically or horizontally. To make a vertical ScrollBar, simply resize the ScrollBar object so that the height is greater than the width.

Although you can resize a ScrollBar in the direction that the thumb travels, user interface guidelines suggest that scrollbars should be no more than 16 pixels thick. This is the default size in REALbasic.

Figure 108. Horizontal and vertical ScrollBars.

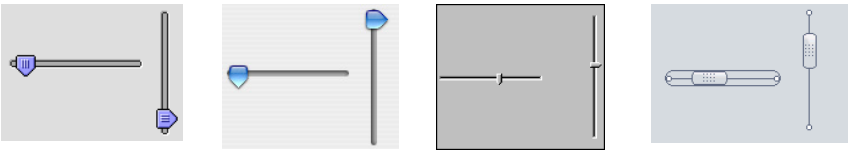


ScrollBars can get the focus on Windows only and have Windows-only properties that you can use to determine when a ScrollBar gets and loses the focus.

Slider

The Slider has the same functionality as a ScrollBar control. However, ScrollBar controls have come to be associated with scrolling text or a picture and less with assigning numeric values. The Slider control provides an interface that is clearly for increasing or decreasing a numeric value. Like the ScrollBar, the Slider control can appear horizontally (which is the default) or vertically. You can create a vertical Slider by changing its height so that it is greater than its width. Unlike the ScrollBar control, the Slider control automatically maintains the correct proportions regardless of the dimensions you give it.

Figure 109. Horizontal and vertical Slider controls.



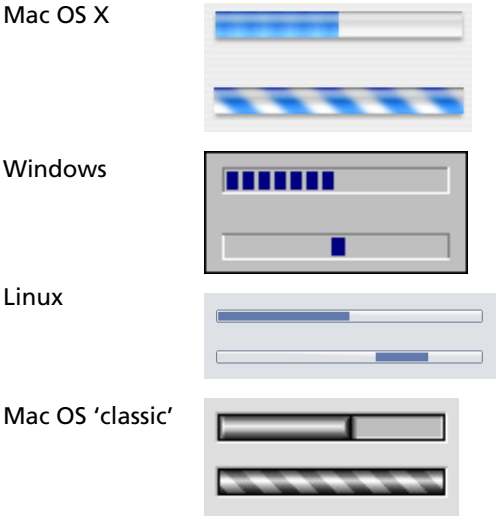
Sliders can get the focus on Linux and Windows. For those platforms, you can use events to determine when a Slider gets and loses the focus.

ProgressBar

ProgressBars are designed to indicate that some function of your application is progressing (hence the name) towards its goal or to show capacity. Unlike ScrollBars and Sliders, ProgressBars are designed to display a value. They cannot be used for data entry. Also, they appear only in a horizontal orientation. When using a ProgressBar to show duration, the ProgressBar can be configured to show progress where the length is determinate or indeterminate. Indeterminate ProgressBars are sometimes referred to as “Barber Poles.”

On Windows and Linux, an indeterminate progress bar takes the form of a single block that moves back and forth.

Figure 110. Determinate and indeterminate ProgressBars.



Controls for Presenting a List of Choices

RadioButton and CheckBox controls can, of course, be used to provide the user with a limited list of choices. There are situations, however, when using these controls is either an inefficient use of space or impossible. Some of these situations are:

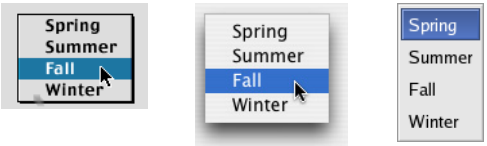
- When the number of choice items is quite long, making it difficult or impossible to use RadioButton or CheckBox controls,
- When the choices change dynamically based on the application's logic,
- When the choice items need to display more than one column of information.

If your situation doesn't match one of these cases, consider using RadioButton or CheckBox controls. They are easier for a new computer user to use because all of their choices will be right in front of them.

ContextualMenu

ContextualMenu controls display a list of choices in a menu when the user right+clicks (Control-clicks on Macintosh) on any control or window that receives a MouseDown event. One ContextualMenu control can actually display contextual menus for several visible controls in the window.

Figure 111. An example of a contextual menu (Windows, Macintosh, and Linux).



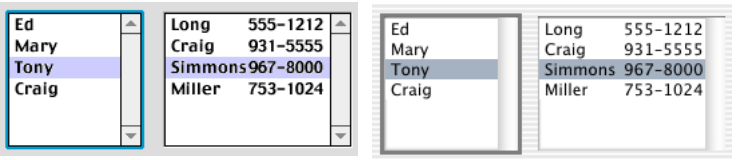
ListBox

ListBox controls display a scrolling list of values. The user can use the mouse or the arrow keys to choose an item. ListBox controls can contain one or more columns of data, can be hierarchical, and can allow one row selection or multiple row selection.

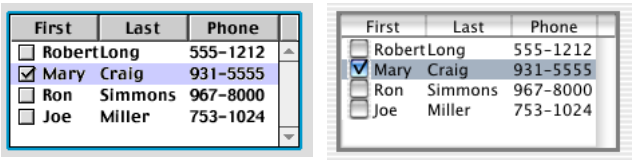
However, you can change the number of columns of any ListBox simply by setting the ColumnCount property in the Properties pane. You can use either icon and modify the number of columns after adding the ListBox to a window.

Figure 112. Examples of ListBoxes.

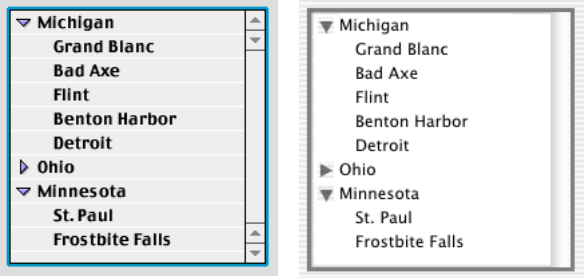
Single column and multiple column



Multiple column with headers and CheckBoxes

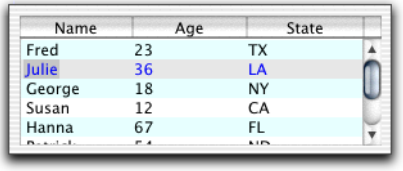


Two column hierarchical



You can also shade cells, rows, and columns programmatically. For example, in Figure 113, alternating rows are shaded differently and the selected row has a custom shading to indicate that it is selected.

Figure 113. An example of custom shading.

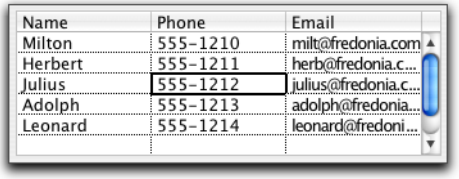


You can also customize row and column borders. Figure 114 on page 119 illustrates the four types of horizontal and vertical rules that can be drawn programmatically. The Default style is “None.”

Figure 114. Four styles of Custom borders.

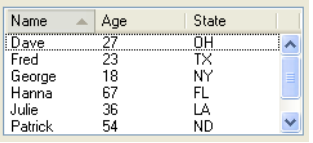
You can also apply these ruling styles to selected cells or even selected cell borders. For example, in Figure 115, a cell containing a phone number is highlighted using Thick Solid borders while the remainder of the ListBox uses Thin Dotted rules:

Figure 115. A custom border around a selected cell.



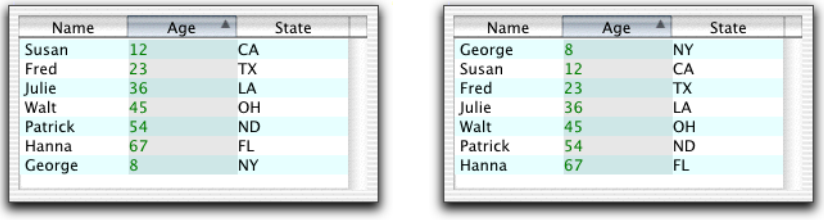
When you can add a header with column labels to a ListBox, the user can sort the data in the ListBox by clicking on a column header or you can sort the rows of the ListBox programmatically. The sort direction is indicated by a sort direction arrow in the header area.

Figure 116. A ListBox sorted by the Name column.



The default sorting method works for alphabetic values, but does not produce valid results for numbers and dates. If you need to sort these data types, you can do so using the CompareRows event of the ListBox. In the following example, code that compares the values of adjacent rows in the Age column is used to get the desired sort order.

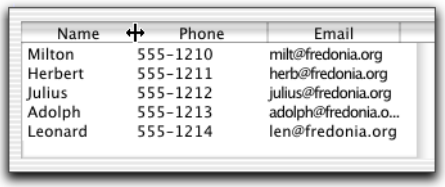
Figure 117. Default and custom sorting on a number column.



Default Sort Custom Sort

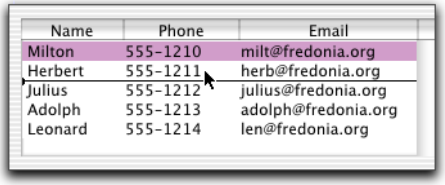
For multicolumn ListBoxes, you can control whether the user can resize columns by dragging column borders. You can enable column resizing on a column-by-column basis or for the entire ListBox. If user resizing is enabled, you can also specify minimum and maximum column sizes. If resizing is enabled, the pointer turns to a column resizing pointer when it is moved to a column border. For example, in Figure 118 the Name column is being resized to make more room for the Email column.

Figure 118. A column being resized.



For both single and multicolumn ListBoxes, you can allow the user to drag columns to rearrange the rows. When you drag, the target for the drop is indicated by a solid line between rows. For example, in Figure 119 the row for Milton is being dropped between Herbert and Julius.

Figure 119. Dragging a row in a ListBox.

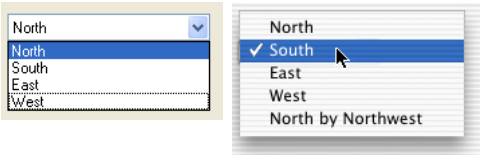


PopupMenu

PopupMenu controls are useful when you have a single column of data to present in a limited amount of space. It presents a list of items and the user can choose one item. When the user displays the PopupMenu's items, the selected item is indicated by a checkmark.

Figure 120. A PopupMenu control.

IDE



Built application

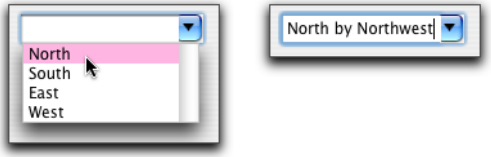
You populate the list by entering them in the InitialValue property in the Properties pane or by calling the AddRow method before the PopupMenu is displayed, for example, in its Open event handler. You can read the value the user has selected (or change the value via code) with the ListIndex property.

ComboBox

A ComboBox combines the features of a PopupMenu and a single-line EditField. The user can either choose an item on the list or enter a different item. Of course, they should be used instead of PopupMenus when you want to allow the user to choose an item that is not on the list.

Figure 121. A ComboBox control.

IDE



The user can choose an item on the list or type an item.

You can populate the list the same way as with a PopupMenu control. The ListIndex property indicates which item the user has selected, but it does not change if the user has typed a value into the ComboBox. Read the value of the Text property to get the current text, which can be either an item on the menu or text entered by the user.

The user can use the down arrow key to display the list when text (or nothing) is displayed in the ComboBox. When the ComboBox has the focus, the user can change the selected item on the list using the up and down arrow keys. The Return key selects the highlighted item in the list and the Escape key closes the popup menu without selecting the highlighted item on the list.

BevelButton

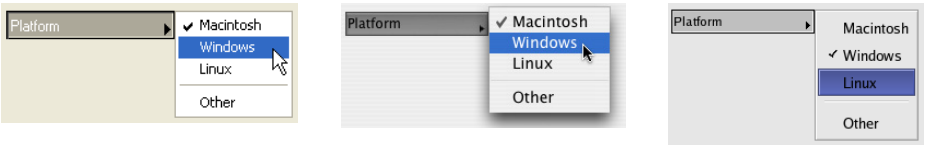
A BevelButton control can be configured to operate as a pop-up menu. Simply set the HasMenu property to 1 or 2 (Normal menu or Menu on Right). See the entry for the BevelButton control in the Language Reference for the list of a BevelButton's properties.

The BevelButton menu shown in Figure 122 was created with this code in the Open event of the BevelButton:

```
me.captionalign=0 //caption aligned flush left
me.hasMenu=2      //menu on right
me.caption="Platform"
me.addRow("Macintosh")
me.addRow("Windows")
me.addRow("Linux")
me.addseparator
me.addRow("Other")
```

You would use the MenuValue property to determine which menu item the user has selected.

Figure 122. A BevelButton popup menu.



Controls for Visually Grouping Other Controls

If a window contains groups of controls in which each group of controls serves a different purpose, it can be confusing to the user to see all of these groups lumped together in a window. It often makes sense (and is sometimes necessary) to group related controls. Fortunately, REALbasic provides several built-in controls to make grouping controls simple.

Separator

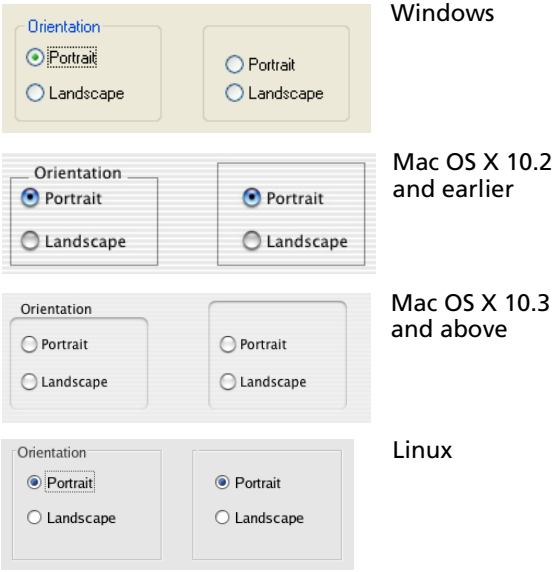
The Separator control simply places a vertical or horizontal line in the window that you can use to help organize other objects.

Figure 123. A Separator control

GroupBox

A GroupBox can be displayed with or without a caption. If a window has more than one group of RadioButton controls, one of the groups must be contained within a GroupBox control in order for the RadioButton groups to function independently. In Mac OS X 10.3 Apple gave GroupBoxes a 3D “sunken” look.

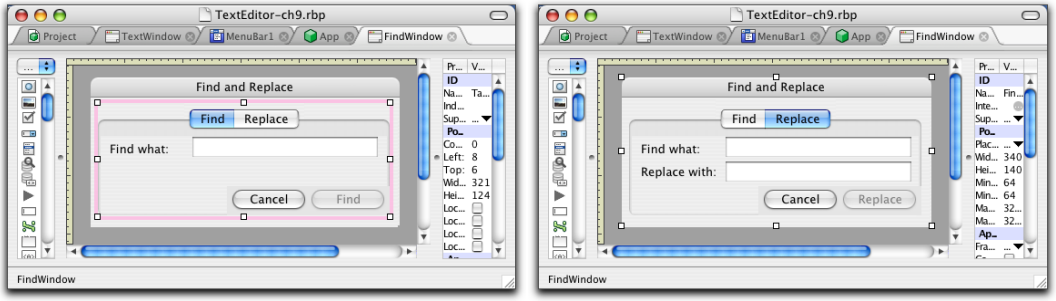
Figure 124. GroupBox controls with and without a caption.



TabPanel

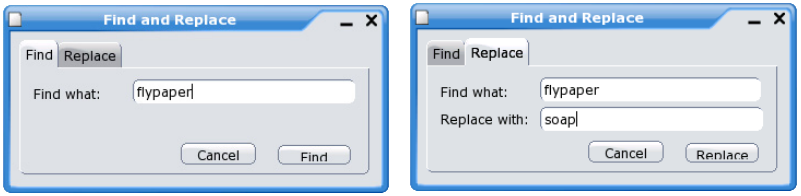
When you have several groups of controls and space is very limited, TabPanels are appropriate. A TabPanel presents each group of controls in a separate panel. When the user clicks on a tab in the TabPanel, REALbasic automatically hides the controls on the current panel and displays the controls on the panel the user selected. In Figure 125, both "Find" and "Find and Replace" functionality is built into the same dialog box using a TabPanel control.

Figure 125. A two-panel TabPanel control.



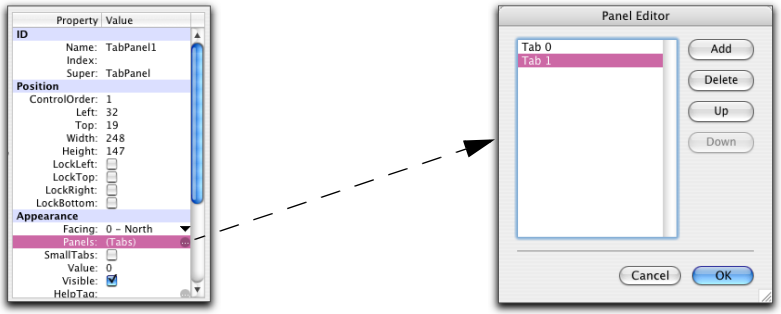
In the built application, the control looks like this:

Figure 126. The two-panel TabPanel control (Linux).



By default, a TabPanel control has two panels. You add, modify, rearrange, or delete tabs and tab labels using the Tab Panel Editor. Click the value of the Panels property of the TabPanel to display the Tab Panel editor.

Figure 127. Opening the Tab Panel Editor.



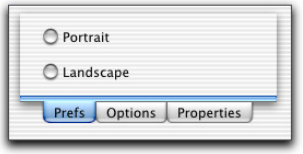
With the Tab Panel Editor, you can:

- **Rename a tab:** Click once on its name to select it and then click again to get an insertion point. When you get the insertion point, replace the current label with the new label.
- **Add a new tab:** Click the Add button. A new tab appears in the list with a default name. The default name is selected, so you can rename it by typing.
- **Delete a tab:** Click once on it to select in and then click the Delete button.
- **Rearrange the tabs:** Highlight a tab you want to move and then click the Up or Down buttons.

  After you have added the desired tabs to the TabPanel, you can add other controls to each page. Click on a tab in the Window Editor and then drag the necessary controls to that page. Repeat the process for each page.

  With the Facings property, you can place the tabs along any edge of the control. Figure 128 shows the use of the "South" setting of the Facings property:

Figure 128. A Tab Panel that uses the "South" Facing.



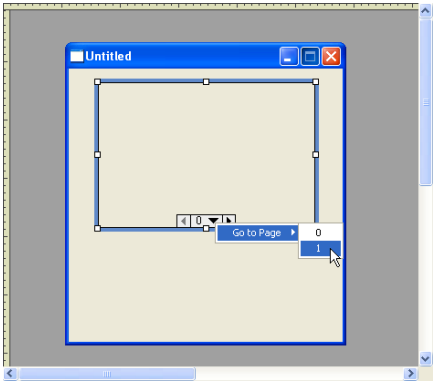
PagePanel

The PagePanel control implements the same idea as the TabPanel, except that the PagePanel control itself is not visible to the end-user. It has no tabs or other navigation widgets, nor does it have a visible border. Only the controls on each page are visible. You are responsible for providing the method of navigating from one page to another. You can do so by setting the value of its Value property programmatically.

In the Development environment, you append, insert, delete, and navigate among pages in a PagePanel control using a widgets at the bottom of the control. This is shown in Figure 129.

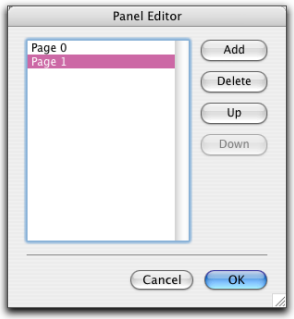
There are two ways to navigate among existing pages. The Go to Page command displays a submenu of page numbers; select a page number from this menu to go directly to a page. Click in the center of the navigation widget to get the drop-down list of page numbers. Or, click the left or right arrows to the left or right of the page number to go to the next or previous page.

Figure 129. The PagePanel's pop-up menu (IDE).



You add, delete, and reorder pages using the PagePanel Editor, shown in Figure 130. You can display the PagePanel editor by clicking the Panels property of the PagePanel in the Properties pane. This is identical to the process illustrated in Figure 127 on page 124 for the TabPanel. The PagePanel editor uses the same interface as the TabPanel editor, except that you cannot name PagePanels. With that exception, the buttons in the PagePanel editor work as described for the TabPanel.

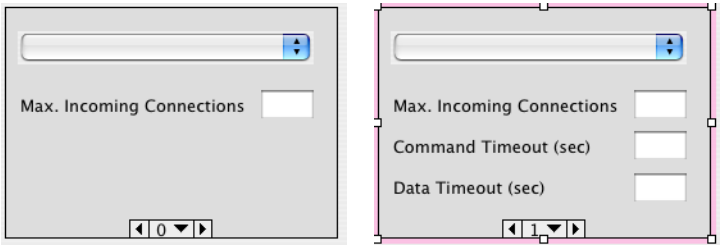
Figure 130. The PagePanel editor.



Here is a simple example of the use of a PagePanel control. The user wants to present two versions of an interface—the 'Basic' version, which shown by default—and an Advanced version, which is shown only if the user chooses to reveal it. A PopupMenu control is used as the widget that controls which page is shown.

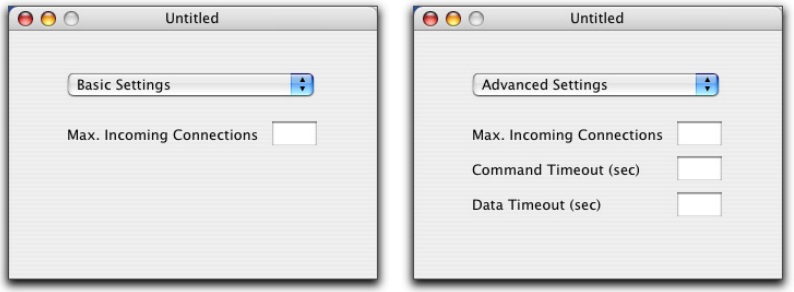
Figure 131 shows the two pages in the IDE.

Figure 131. The 'Basic' and 'Advanced' pages in the IDE.



In the built application, the two pages look like this.

Figure 132. The Basic and Advanced settings pages in the built application.



Code in the Change event of each pop-up menu controls the page that is displayed and the setting of the pop-up menu on the other page. For example, the code for the pop-up menu on the 'Basic' page is:

```
pagePanel1.value=me.listindex
popupMenu2.listindex=me.listindex
```

Controls for Displaying Graphics and Pictures

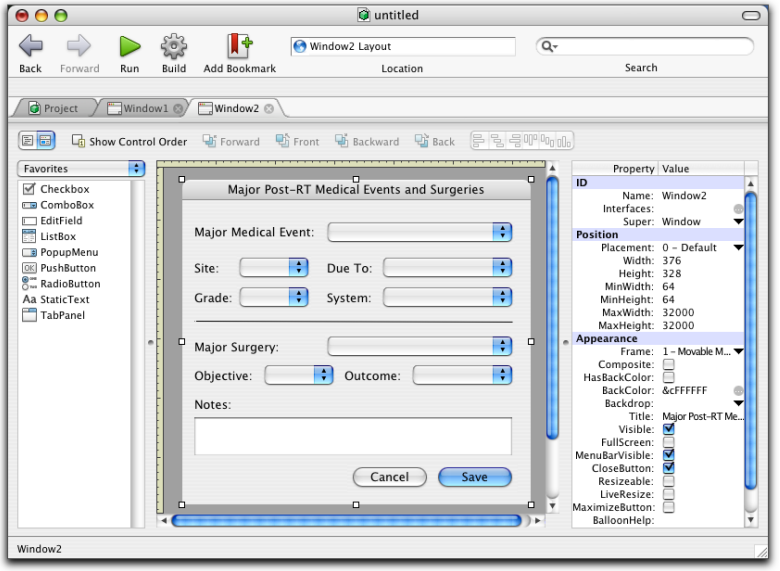
REALbasic is very flexible when it comes to displaying graphics and pictures. You can use the built-in graphic controls, display pictures from documents, or draw the graphics using REALbasic's programming language.

Line

A Line control draws a line that can be of any length, width, color, and direction. By default, lines are 100 pixels in length, 1 pixel in width, black, and horizontal.

In Figure 133, a Line control is used to divide two areas of a complex dialog box used for data input.

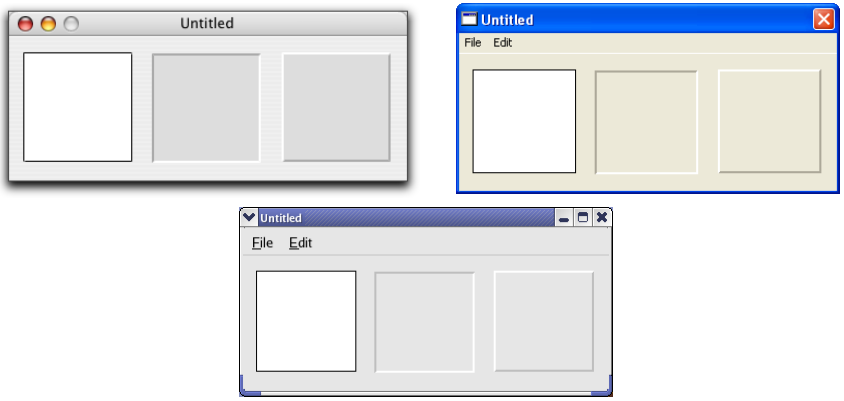
Figure 133. A Line control used to divide two sections in a dialog box.



Rectangle

A Rectangle control draws a rectangle that can be of any length, width, border color, and fill color. By default, rectangles are 100 pixels in length and width, with a black border that is 1 pixel thick and a white center. Because you can control the color of the left and top borders independently from the right and bottom borders, you can easily create rectangles that appear to be sunken or raised. The example code is in the entry for the Rectangle control in the *Language Reference*.

Figure 134. A Rectangle with default, sunken and raised appearances.



RoundRectangle

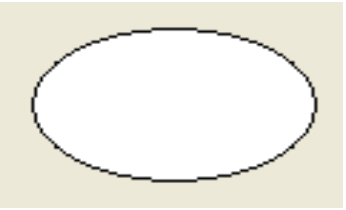
RoundRectangles are similar to regular Rectangle controls. The differences are that you don't have the independent color control for the border (because it is one continuous line) but you can control the width and height of the arcs that make up the round corners.

Figure 135. A RoundRectangle control.

Oval

Draws an oval with a single pixel, black border, and filled with white. All of these properties can be modified. The "ovalness" of the Oval is controlled by its height and width. For example, an Oval with the same width and height is a perfect circle.

Figure 136. An Oval control.



Canvas

A Canvas control can be used to display a picture from a file or a picture drawn using REALbasic's programming language. If your application requires a type of control that is not built-in, you can use a Canvas control and REALbasic drawing commands to create the controls you need. The Canvas control has access to the drawing tools belonging to the Graphics class; with these tools you can programmatically draw objects within the Canvas.

The Canvas control can get the focus, so you can emulate any other type of control that you would like to get the focus.

Canvas controls can be used to create extremely sophisticated controls. In Figure 137, a Canvas control is used to provide a table of data with rows that can be selected and columns that can be sorted by clicking on the column title.

Figure 137. A sophisticated control created using a Canvas control.

Name	Age	City
Smith	20	London
Jones	10	Paris
Blake	30	Paris
Clark	20	London
Adams	30	Athens
Björn	22	Reykjavik

The Table control above was created by Björn Eiríksson.

ImageWell

The ImageWell control provides an area in which you can display a BMP or PNG image on Windows and Linux or a PICT image on Macintosh. You can easily program the ImageWell control to accept a dragged picture.

Figure 138. An ImageWell.

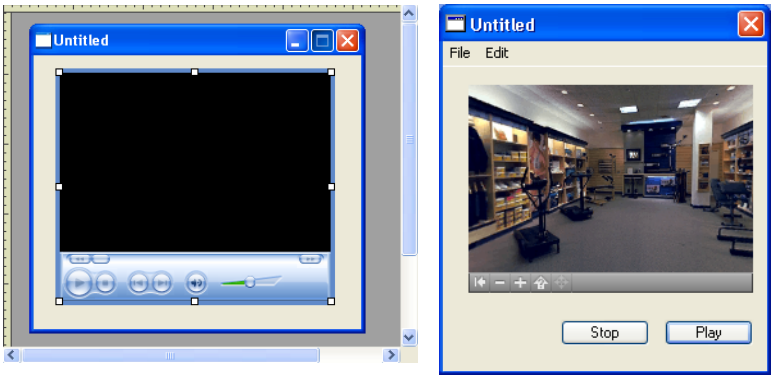
Controls for Playing Movies, Music, and Animation

If the user has QuickTime™ or the Windows Media Player installed, your application can play movies with the MoviePlayer control. If it has QuickTime Musical Instruments, it can use the NotePlayer control to play music.

MoviePlayer

The MoviePlayer control displays the standard QuickTime or Windows Media Player movie controller, depending on your platform. On Linux, neither player is supported.

Figure 139. A MoviePlayer control in the IDE and an application (QT3D movie shown).



From the IDE, you can select the movie that will be associated with a particular MoviePlayer control. You can also determine the default appearance of the movie controller. Your choices are: the controller is displayed, a badge (a small icon that, when clicked, reveals the controller) is displayed, or no controls are displayed.

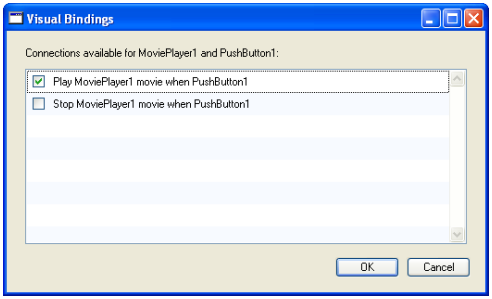
Assigning a movie to a MoviePlayer control is amazingly easy. First, add the movie to the Project Editor. You can either drag it from the desktop to the Project Editor or use the File ▶ Import menu command.

When you add a movie to the Project Editor, it automatically appears as an item in a MoviePlayer's Movie property pop-up menu. Use the pop-up menu to assign the movie to the Movie property of the control.

You can then add Stop and Play PushButtons to the window and assign them actions using object binding, or let the users use the built in controls in the player.

If you wish to provide your own controls, add a PushButton to the window. Select both the PushButton and the MoviePlayer control and choose Project ▶ Add ▶ Binding. (You may have added the Add Binding command to your Window Editor toolbar; if so, you can add the binding by clicking its button.)

A Visual Bindings dialog box appears, giving you a choice of automatic actions:



Choose Play MoviePlayer1 Movie when PushButton1 pushed. Next, add another PushButton to the window and assign the Stop MoviePlayer1 Movie binding to it. The result is a fully functional MoviePlayer application shown in Figure 139 on page 130.

The MoviePlayer is not currently supported under Linux.

NotePlayer

Although the NotePlayer control displays an icon when placed in a window in the Window Editor, it has no interface. It is designed only for playing musical notes using QuickTime Musical Instruments. See the NotePlayer control in the Language Reference for more details.

The NotePlayer is not currently supported under Linux.

SpriteSurface

This control is used to create animation. When you call the SpriteSurface's Run method, the menu bar, all windows, and the desktop are hidden. This allows the animation to fill the screen. This animation is done using *Sprites*. A Sprite is simply an object with a picture that can be moved across the screen by your programming code. The SpriteSurface will automatically handle all of the drawing of the background and the sprites for you. The SpriteSurface will tell you when two sprites collide and give you the opportunity to test for keys pressed by the user to allow you to you interact with them.

For a complete list of SpriteSurface properties, see the *Language Reference*.

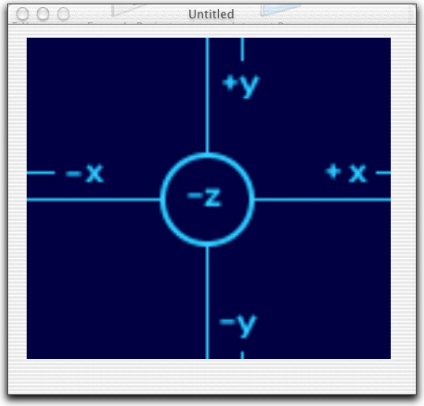
Rb3DSpace

The Rb3DSpace control is used to display animation in a three-dimensional space. It works in conjunction with 3D classes which allow you to load, group, and manipulate 3D objects. The base 3D object is derived from the Object3D class; 3D objects are presented to the user via the Rb3DSpace control.

The Rb3DSpace control requires either the QuickTime 3D or Quesa libraries. If you are building for Mac OS "classic," the QuickTime libraries are installed by default and you don't need to do any extra installation. On Mac OS X, you need to install the Quesa libraries, which are available on the CD or at the REALbasic web site.

To get started, place an Rb3DSpace control in a window and turn on the DebugCube property. Click the Run button. If the 3D libraries are installed, you will see the three-dimensional space with the coordinate axes labeled.

Figure 140. An empty 3D space with the coordinates shown.



The X and Y axes are the horizontal and vertical, respectively and the Z axis is the near-far axis. The "origin" or 0,0,0 point is where the X, Y, an Z axes intersect. If you don't see the coordinates with DebugCube on, chances are the required libraries are not installed on your computer.

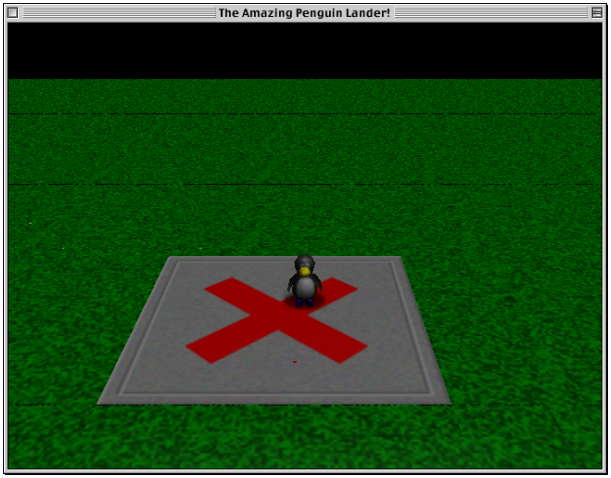
After verifying that the necessary libraries are available, you will want to load an object into the space and move the camera away from the object so you can see it. Check out the examples for the Rb3DSpace control in the *Language Reference* for some tips on loading and animating 3D objects.

The Rb3DSpace control is best illustrated by an example. In Figure 141, the position and flightpath of the Amazing Penguin is manipulated with the mouse in the 3D space.

- **Yaw** or Left-Right movements
- **Pitch** or Up-Down movements
- **Roll** or clockwise-counterclockwise movements

Also, you can move the position of the "camera" in the 3D space.

Figure 141. An example Rb3DSpace control.



See the Example Projects on the CD and the *Language Reference* for more information on the Rb3DSpace control.

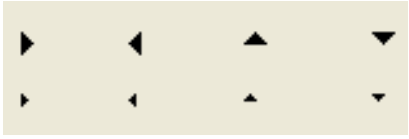
Miscellaneous Controls

PopupArrow Control

The PopupArrow controls places an arrow in the window that points in any of four directions. Two sizes of arrows are available.

You control both the direction and size of the popup arrow using one property, the Facing property. You get a choice of four orientations (North, South, East, and West) and two sizes, standard and small. Typically, you would use a PopupArrow control as part of a custom control. The orientation of the arrow indicates whether the custom control can display additional information or options. Figure 142 shows all four orientations and both sizes.

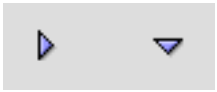
Figure 142. Examples of the PopupArrow Control.



Disclosure Triangle Control

A disclosure triangle control is used to display hierarchical lists, i.e., the List view of files and folders in a Finder window. In REALbasic, you can control the direction of the DisclosureTriangle (left or right) and whether it is in the 'disclosed' (down) state.

Figure 143. Disclosure Triangles.



UpDownArrows Control

The UpDownArrows control is commonly used as an interface for scrolling. You use two events, Up and Down, to determine whether the user has clicked an arrow.

Figure 144. The UpDownArrows Control (Windows, Linux, and Macintosh).

ProgressWheel Control

The ProgressWheel control is often displayed to indicate that a time-consuming operation is in progress. The ProgressWheel control appears when its Visible property is set to True. It is animated automatically

Figure 145. The ProgressWheel Control (Windows, Linux, and Macintosh).

RbScript Control

The RbScript control allows the end user to write and execute REALbasic code within a compiled application. Scripts are compiled into machine code.

You pass the REALbasic code that you want to run via the Source property and execute it by issuing the Run method. Please see the *Language Reference* for details on the functions, control structures, and commands supported by the RbScript control.

You can also automate the IDE itself via RbScripts. Any menu command or toolbar button can be automated. You can also get and set property values, navigate among panels in the IDE, manipulate code in the Code Editor, and automate the build process among other things. To script the IDE, you don't use the RbScript control. Instead, use the File ▶ IDE Scripts ▶ New IDE Script command to open the IDE Script editor. See the entry for RBScript in the *Language Reference* for the commands used in IDE scripting.

Controls for Handling Communications

REALbasic provides controls that allow your application to communicate through the serial port (for communicating via a modem or through a serial cable to another device) and over a network to other computers using TCP/IP, the Internet's communication protocol.

Serial

Although the Serial control displays an icon when placed in a window in the Window Editor, it is not visible in the built application. It is designed only for execut-

ing code to communicate via the serial port. For more information, see the Serial control in the *Language Reference* for more details.

The Serial control can be instantiated via code since it is not a subclass of Control. This allows you to easily write code that does communications without adding the control to a window.

TCPSocket

Although the TCPSocket control displays an icon when placed in a window in the Window Editor, it has no interface. It is designed only for executing code to communicate with other computers on the Intranet or Internet using TCP/IP.

The TCPSocket control can be instantiated via code since it is not a subclass of Control. This allows you to easily write code that does communications without adding the control to a window.

For more information, see the section on the TCPSocket control in the Language Reference.

SSLSocket

The SSLSocket control is similar to the TCPSocket. The SSLSocket implements Secure Sockets Layer communication via TCP/IP. It supports SSL versions 2 and 3 as well as TLS (Transport Layer Security) version 1.

However, it does not have an icon of its own in the Controls pane. But since it is not derived from the Control class, you can create an SSLSocket via code. You can also add it to a window via the window's contextual menu.

To establish an SSL connection with the SSLSocket, set the Secure property to True and use the Connect method.

The SSLSocket is a Professional-only feature of REALbasic; the Standard version of REALbasic does not compile references to the SSLSocket.

For more information, see the section on the SSLSocket control in the Language Reference.

ServerSocket

The ServerSocket class enables you to support multiple TCP/IP connections on the same port. When a connection is made on that port, the ServerSocket hands the connection off to another socket, and continues listening on the same port. It includes the ability to replenish its supply of TCPSockets as connections are made. Without the ServerSocket, it is difficult to implement this functionality due to the latency between a connection coming in, being handed off, creating a new listening socket, and restarting the listening process. If you had two connections coming in at about the same time, one of the connections may be dropped because there was no listening socket available on that port.

For more information, see the section on the ServerSocket class in the *Language Reference*.

UDPSocket

The UDPSocket supports communications via a UDP (User Datagram Protocol) connection. It also can be created via code because it is not derived from the Control class.

UDP is the basis for most high speed, highly distributed network traffic. It is a connectionless protocol that has very low overhead, but is not as secure as TCP. Since there is no connection, you do not need to take nearly as many steps to prepare when you wish to use a UDP socket.

UDP sockets can operate in various modes, which are all very similar, but have vastly different uses. Perhaps the most common use is "multicasting." Multicasting is a lot like a chat room: you enter the chatroom, and are able to hold conversations with everyone else in the chatroom.

For more information, see the section on the UDPSocket class in the *Language Reference*.

"Easy" Communications

The REALbasic language includes "easy" versions of the UDPSocket and TCPSocket classes that are designed for communication among REALbasic applications on a network. They are called the EasyUDPSocket and EasyTCPSocket classes. Also, AutoDiscovery class can automatically poll the network and return the IP addresses of the computers that are logged in for "easy" communications. Once this is done, it is very easy for members to send messages to one another.

These classes make it easy to set up a network of REALbasic users, but they are not designed for more generic communication with other applications, such as an FTP or HTTP server.

For more information on the "Easy" communication classes, see the section "Making Networking Easy" on page 533 and the entries for the classes in the Language Reference.

ToolbarItem and StandardToolbarItem

The ToolbarItem and StandardToolbarItem controls enable you to create a toolbar in a window on your Mac OS X applications. Using the ToolbarItem control, you can populate your toolbar using standard images provide by Apple Computer, Inc. or create and reference your own icons.

For more information, see the sections on the ToolbarItem and StandardToolbarItem controls in the *Language Reference*.

The Timer

The Timer executes some code once or repeatedly after a period of time has passed. Although the Timer displays an icon when placed in a window in the Window Editor, it is not visible in built applications. It is not a control and can be created with code. See the section on the Timer object in the *Language Reference* for more details.

Controls for Working With Databases

There are two special-purpose controls that are relevant only in projects that access databases: the DatabaseQuery control and the DataControl control. For more information about both controls, see Chapter 10.

DatabaseQuery

The DatabaseQuery control can be used to send SQL queries to the database. This function can also be done with the language (without using the DatabaseQuery control at all). When you add a DatabaseQuery control to a window, it is not visible in the built application.

To use the DatabaseQuery control, you write a SQL statement and assign it to the SQLQuery property of the control. The SQLQuery is executed automatically when its window appears. It also has one method, RunQuery, which can be used to run SQLQuery.

See the example of object binding using the DatabaseQuery control in "Using Object Binding with a Database Query" on page 489.

DataControl

The DataControl control is a "composite" control that provides a very easy way to build a front-end interface to a database. It consists of four record navigation buttons (First Record, Previous Record, Next Record, and Last Record) and a caption.

Figure 146. A DataControl control used for navigating among records.

When linked to a database and controls that display data, you can create a fully-functional database front-end with no programming. See the example in the section "The DataControl Control" on page 490.

The Spotlight Query Control

Mac OS X 10.4 includes a sophisticated search engine known as "Spotlight." It relies on metadata attributes of items to do its searches. The SpotlightQuery class in the REALbasic language provides access to the Apple Spotlight API so that you can incorporate Spotlight searches in your REALbasic applications. You can pass queries to Spotlight, determine when it has results, and then display those results in standard REALbasic controls such as ListBoxes and EditFields.

Of course, these capabilities are supported only for users who are running Mac OS X 10.4 and above. On all other operating systems, the SpotlightQuery control and your Spotlight-related code will do nothing.

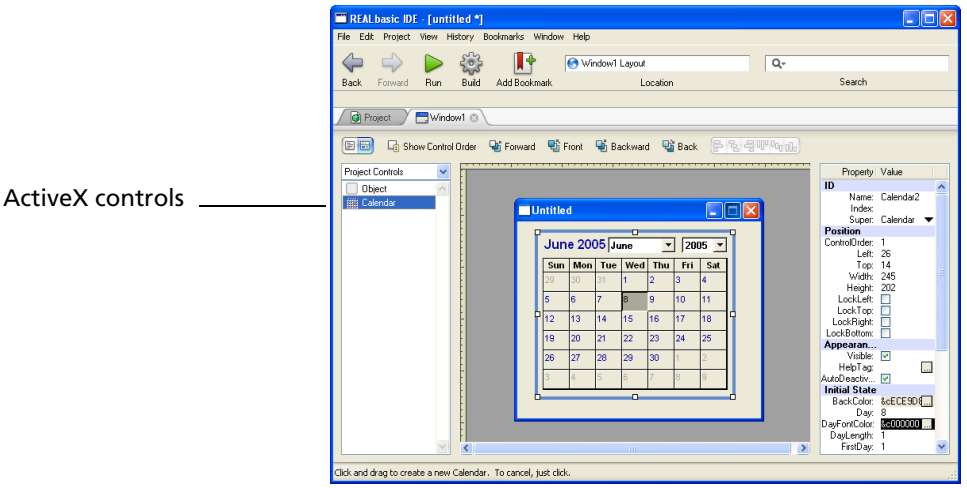
You use the SpotlightQuery class to perform these searches. Use the Query property of this class to pass your Spotlight query to the API. The SpotlightQuery class is listed in the Built-in Controls list to make it convenient to add it to a window, but it is not subclassed from the Control class. You can also create SpotlightQuery objects via the language and implement your Spotlight support that way.

The entry for SpotlightQuery in the Language Reference has examples of Spotlight support both with and without a SpotlightQuery control in a window.

ActiveX Controls

The Windows version of REALbasic allows you to add ActiveX controls to the Controls pane. They are added as Project controls. Once added to the Controls pane, an ActiveX control can be added to your application as if it were a part of REALbasic.

Figure 147. An ActiveX Calendar control in the Project Controls list.

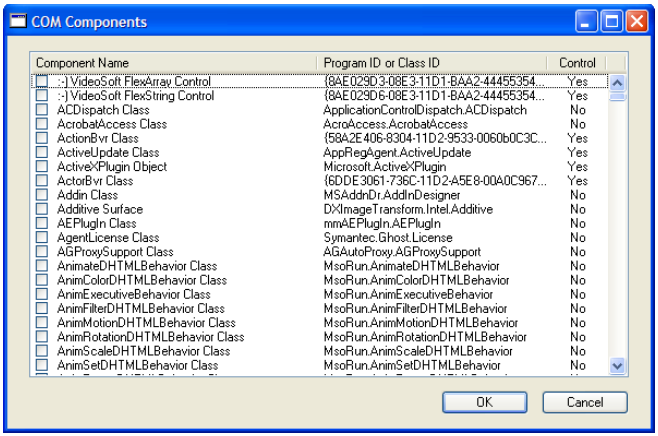


To add an ActiveX control, do this:

1 **In the Windows version of REALbasic, click the Add ActiveX Component button in the Project Editor toolbar or choose Project ▸ Add ▸ ActiveX Component.**

The ActiveX Components dialog appears, listing all the ActiveX controls and objects that are currently installed on your computer.

Figure 148. The ActiveX Components dialog box.



2 Select the controls or programmable objects that you wish to add to your Project Controls list and click OK.

The selected controls or objects are added to the items in the Project Editor. If they are ActiveX controls, they are also added to the list of Project controls in the all the Window Editors in your project. Controls generally have their own icon, while programmable objects use the icon for the OLE Container control.

The ActiveX components that you add in this manner become part of the current project rather than your copy of REALbasic. To add the ActiveX components to other projects, repeat this process.

You add an ActiveX control to a window just as any built-in control. Drag it to a window and use the Code Editor to add code to the ActiveX control. Please refer to Microsoft documentation for information about programming ActiveX components and the specific ActiveX components you want to use. Microsoft documentation is in the MSDN library at:

```
http://msdn.microsoft.com/library/
```

OLEContainer Control

The OLEContainer control enables you to embed ActiveX controls in your interface. ActiveX is a Windows-only feature. The ActiveX controls that you add to your project are derived from OLEContainer class.

The Container Control

The Container Control is a special type of control that can contain all other types of controls and other Container Controls. However, the Container Control itself is invisible in built applications. Think of it as a blank canvas onto which you can place visible controls. It enables you to organize several controls into groups and manage dynamic layouts. ContainerControls have multiple uses, You can:

- Organize groups of controls into reusable interface components,

- Create custom controls made up of several constituent controls,
- Increase encapsulation of complex window layouts,
- Create dynamic layouts.

Unlike all other controls, the Container Control does not appear in the list of controls in the Window Editor. It appears instead in the Project Editor toolbar or in the Project ▶ Add submenu.

To use a Container Control, add a Container Control to the Project Editor. Double-click it to open its Container Control Editor. Its editor looks exactly like a Window Editor except that the Container Control is represented by a rectangle. A Container Control has no frame and no window widgets, such as the Minimize, Maximize, and Close buttons of a document window.

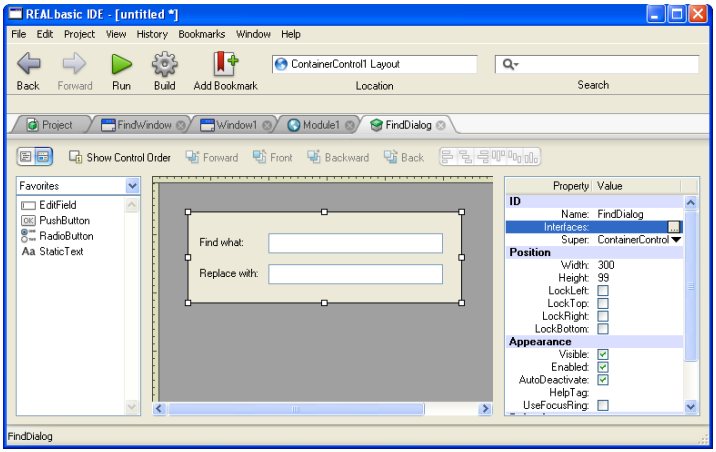
You place controls in a Container Control just as if it were a window in a Window Editor. Controls that are placed in a Container Control have a control order that works inside the parent Container Control.

To use a Container Control in your project, follow these general steps:

- Add a Container Control to your project by clicking the Add Container Control button in the Project Editor toolbar or with the Project ▶ Add ▶ Container Control menu item,
- Rename it using its Properties pane (optional),
- Double-click the Container Control in the Project Editor to open its editor and then design the instance of the Container Control in its Container Control Editor,
- Display the window in which you want to display the Container Control in its Window Editor. Use the Control pane's drop-down list to switch to Project Controls. Your Container Controls will be listed in the Project Controls, not the Built-in Controls.
- Add the Container Control to the window.

Figure 149 is an example Container Control in its editor that will be used in a Find/Replace dialog box. It consists of two StaticText controls and two EditFields.

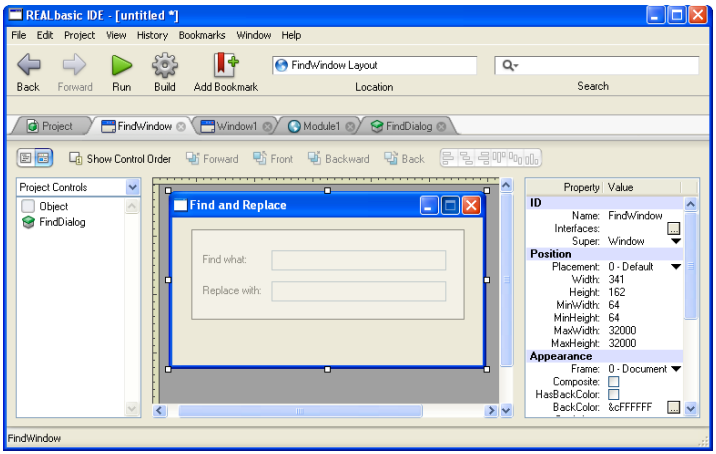
Figure 149. The Container Control in its Container Control Editor.



You can also place and position it via the language. This feature enables you to create dynamic screens.

In the Figure 150, the Container Control shown in Figure 149 has been placed inside a TabPanel control in a Find and Replace dialog box.

Figure 150. A Container Control in a window.



Object Binding

Once you have added the application's controls to a window, you usually use the Code Editor to add any desired functionality. For example, you can use the Code Editor to specify the behavior of a PushButton when the user clicks it by adding some code to the PushButton's Action event.

In some cases, you can actually add functionality without writing any code whatsoever. To do this you use a feature called "object binding."

With object binding, you specify an action when some aspect of one control changes without code. A simple example of object binding was presented in the section on the MoviePlayer control on page 130. Two PushButtons, a "Play" and a "Stop" PushButton were bound to the MoviePlayer control. The bindings themselves specify the functionality; there is no code being written "behind the scenes."

In the case of the MoviePlayer example, the binding has a directional characteristic. One control is referred to as the "source" control—the control that the user interacts with—and the other control is the "destination" control—the control that does something when the user invokes the binding

To establish an object binding, do this:

1 Select the two controls for which you want to establish a binding and choose Project ▸ Add ▸ Binding or click the Add Binding button in the toolbar, if it is available.

The Visual Bindings dialog box appears, listing all the appropriate bindings for the controls you selected. The built-in bindings are shown in Table 1 on page 142. Custom bindings can be added.

The Add Binding and List Bindings commands are optional commands that can be added to the Window Editor toolbar. If they have been added, you can use them instead of using the corresponding menu commands.

2 Choose the desired action and click OK.

A line connecting the two controls appears in the window. When you run your application, the binding works just as if you had entered equivalent code for the action in the Code Editor. For example, the object bindings in the MoviePlayer example are equivalent to the lines MoviePlayer1.Play and MoviePlayer1.Stop that could have been inserted into the Action events of the Play and Stop buttons, respectively.

To delete an existing object bindings, do this:

1 If available, click the List Bindings button in the toolbar or choose View ▸ List Bindings.

A dialog box that lists all the object bindings for that window appears.

2 Click on the binding you wish to remove and click the Delete button.

The object binding that you specified is removed and the line connecting the objects disappears. Of course, you can use the Add Binding command to establish a different type of binding.

Here are the binds that are included with REALbasic.

Table 1: Built-in object Binds in REALbasic

Source Object	Target Object	Binds
PushButton or BevelButton	MoviePlayer	Play MoviePlayer when Button is pushed Stop MoviePlayer when Button is pushed.

Table 1: Built-in object Binds in REALbasic (Continued)

Source Object	Target Object	Binds
PushButton or BevelButton	DataBaseQuery	Requery DatabaseQuery when button is pushed.
PushButton or BevelButton	ListBox	Enable Button when ListBox has a selection.
RadioButton	DatabaseQuery	Requery DatabaseQuery when RadioButton is selected. Requery DatabaseQuery when RadioButton is deselected.
PushButton	DataControl	Insert Record when PushButton is pushed. Update Record when PushButton is pushed. Delete Record when PushButton is pushed. New Record when PushButton is pushed.
BevelButton	DataControl	Insert Record when BevelButton is pressed. Update Record when BevelButton is pressed. Delete Record when BevelButton is pressed. New Record when BevelButton is pressed.
RadioButton	DataControl	Insert Record when RadioButton is selected. Update Record when RadioButton is selected. Delete Record when RadioButton is selected. New Record when RadioButton is selected. Insert Record when RadioButton is deselected. Update Record when RadioButton is deselected. Delete Record when RadioButton is deselected. New Record when RadioButton is deselected.
CheckBox	DataControl	Insert Record when CheckBox is checked. Update Record when CheckBox is checked. Delete Record when CheckBox is checked. New Record when CheckBox is checked. Insert Record when CheckBox is unchecked. Update Record when CheckBox is unchecked. Delete Record when CheckBox is unchecked. New Record when CheckBox is unchecked. Enable DataControl when CheckBox is checked.
RadioButton	ListBox	Enable RadioButton when ListBox has a selection.
RadioButton	MoviePlayer	Play MoviePlayer movie when RadioButton is selected. Stop MoviePlayer movie when RadioButton is selected. Play MoviePlayer movie when RadioButton is deselected. Stop MoviePlayer movie when RadioButton is deselected.

Table 1: Built-in object Binds in REALbasic (Continued)

Source Object	Target Object	Binds
DatabaseQuery	PopupMenu	Bind PopupMenu with DatabaseQuery results. Bind DatabaseQuery parameter with string data from PopupMenu. Bind DatabaseQuery parameter with string data from PopupMenu RowTag. (See the section "Using Object Binding with a Database Query" on page 489 for an example of these bindings.)
DatabaseQuery	ListBox	Bind ListBox with list data from DatabaseQuery results. Bind DatabaseQuery parameter with string data from ListBox. Requery DatabaseQuery when ListBox gains focus. Requery DatabaseQuery when ListBox loses focus.
CheckBox	NotePlayer	Enable NotePlayer when CheckBox checked.
CheckBox	DatabaseQuery	Requery DatabaseQuery when CheckBox is checked. Requery DatabaseQuery when CheckBox is unchecked.
CheckBox	Contextual Menu	Enable ContextualMenu when CheckBox is checked.
CheckBox	PopupMenu	Enable PopupMenu when CheckBox is checked.
CheckBox	PopupArrow	Enable PopupArrow when CheckBox is checked.
CheckBox	DisclosureTriangle	Enable DisclosureTriangle when CheckBox is checked.
CheckBox	SpriteSurface	Enable SpriteSurface when CheckBox is checked.
CheckBox	NotePlayer	Enable NotePlayer when CheckBox is checked.
CheckBox	ImageWell	Enable ImageWell when CheckBox is checked.
CheckBox	TabPanel	Enable TabPanel when CheckBox is checked.
CheckBox	ProgressWheel	Enable ProgressWheel when CheckBox is checked.
CheckBox	ImageWell	Enable ImageWell when CheckBox checked.
CheckBox	ListBox	Enable ListBox when CheckBox is checked. Enable CheckBox when ListBox has a selection.
CheckBox	EditField	Enable EditField when CheckBox is checked.
CheckBox	Canvas	Enable Canvas when CheckBox checked.
CheckBox	ProgressBar	Enable ProgressBar when CheckBox checked.

Table 1: Built-in object Binds in REALbasic (Continued)

Source Object	Target Object	Binds
CheckBox	MoviePlayer	Play MoviePlayer movie when CheckBox is checked. Stop MoviePlayer movie when CheckBox is checked. Play MoviePlayer movie when CheckBox is unchecked. Stop MoviePlayer movie when CheckBox is unchecked. Enable MoviePlayer when CheckBox is checked.
CheckBox	SpriteSurface	Enable SpriteSurface when CheckBox checked.
CheckBox	Slider	Enable Slider when CheckBox is checked.
CheckBox	ScrollBar	Enable ScrollBar when CheckBox is checked.
ListBox	UpDownArrows	Enable UpDownArrows when ListBox has a selection.
ListBox	DisclosureTriangle	Enable DisclosureTriangle when ListBox has a selection.
ListBox	PopupMenu	Enable PopupMenu when ListBox has a selection.
ListBox	PopupArrow	Enable PopupArrow when ListBox has a selection.
ListBox	ProgressWheel	Enable ProgressWheel when ListBox has a selection.
ListBox	ContextualMenu	Enable ContextualMenu when ListBox has a selection.
ListBox	ScrollBar	Enable ScrollBar when ListBox has a selection.
ListBox	NotePlayer	Enable NotePlayer when ListBox has a selection.
ListBox	Slider	Enable Slider when ListBox has a selection.
ListBox	MoviePlayer	Play MoviePlayer movie when ListBox gains focus. Stop MoviePlayer movie when ListBox gains focus. Play MoviePlayer movie when ListBox loses focus. Stop MoviePlayer movie when ListBox loses focus. Enable MoviePlayer when ListBox has a selection.
ListBox	ImageWell	Enable ImageWell when ListBox has a selection.
ListBox	Canvas	Enable Canvas when ListBox has a selection.
ListBox	EditField	Bind EditField with string data from ListBox. (copies selected item in ListBox to EditField) Enable EditField when ListBox has a selection.
ListBox	ProgressBar	Enable ProgressBar when ListBox has a selection.
ListBox	TabPanel	Enable TabPanel when LIstBox has a selection.
EditField	PopupMenu	Bind EditField with string data from PopupMenu (copies selected menu item into EditField). Bind EditField with string data from PopupMenu RowTag.

In addition, it is possible to define custom object bindings using the language. For information on custom bindings, see the section "Custom Object Bindings" on page 468.

Changing The Tab (Control) Order

The order in which the user moves through controls that receive the focus when he presses the Tab key is called the *Control Order* (also known as the *Tab Order*). The Control Order is actually controlled by the control layers. When a window opens, REALbasic places the focus in the control that is farthest back that can also receive the focus. (A control is said to be in the back when another control can cover it up when it is moved to the same location). You can change the control order in three ways:

- Using the Window Editor toolbar to view and modify the control order,
- Using the equivalent menu commands in the Edit ▶ Arrange submenu,
- Using the Properties Panel.

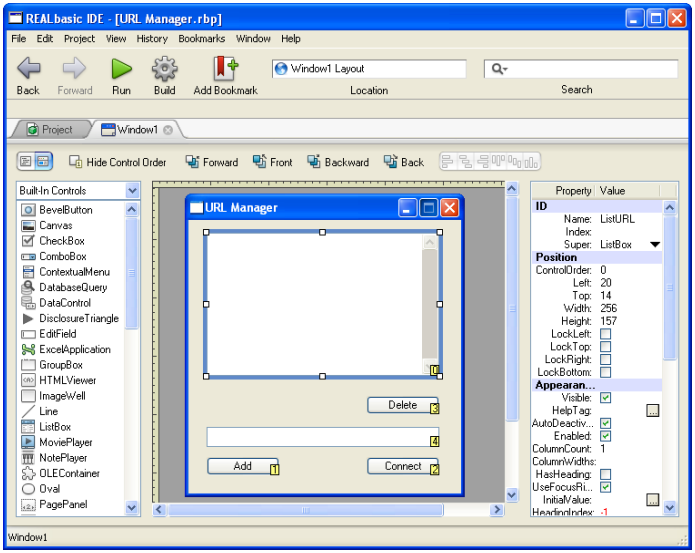
The Window Editor toolbar menu contains five commands that enable you to change the Control Order while working with the window itself. First, you can view the current Control Order by clicking the Show Control Order button in the Window Editor tab or by choosing the View ▶ Control Order menu command. When the Control Order menu command is selected, it has a checkmark to its left. When the Control Order is shown, each control is augmented by a badge that shows its control order number.

The Control Order that is shown assumes that the control is capable of getting the focus. Some controls do not receive the focus on Macintosh, but are included in the shown control order if they can get the focus on any platform on which REALbasic can run. For example, PushButtons, CheckBoxes, and RadioButtons do not get the focus on Macintosh, but are shown in the control order within the Macintosh IDE.

The control order that is shown is valid for Windows and Linux builds of the application. For Macintosh builds, you need to ignore the ControlOrder property of a control that will not receive the focus. The ControlOrder property, however, is valid on Macintosh if the user has the Full Keyboard Access option enabled at the Mac OS X system level.

A Window Editor with the Show Control Order command selected is shown in Figure 151.

Figure 151. A window with Control Order shown.



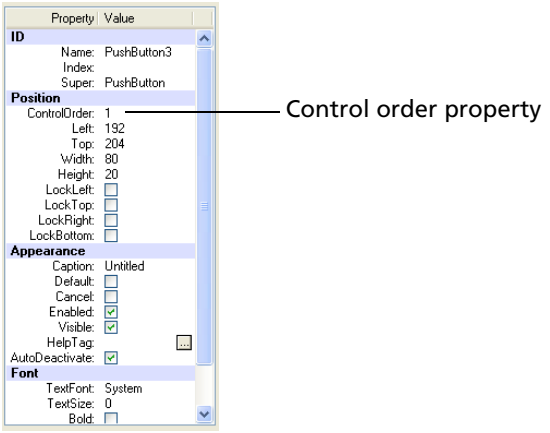
The control with the lowest number is the one that gets the focus first. It's the one in the back. Once you see the Control Order, you can easily modify the order of a control by clicking on the control and choosing one of the following control order buttons:

- Forward
- Front
- Backward
- Back

Front and Back move the control to the top or bottom of the order, while Forward and Backward move it one position up or down. When you click one of these buttons, each control's badge updates to show its new position in the Control Order.

If you don't care for either the Window Editor toolbar or the Edit ▶ Arrange submenu, you can also change the Control Order using each control's Properties pane. The control's current position in the Control Order is listed as the first item in the Position group in the Properties pane. You can change it simply by entering a different value and pressing the Tab or Enter key, as you would change any other property.

Figure 152. A control's ControlOrder in its Properties Pane.



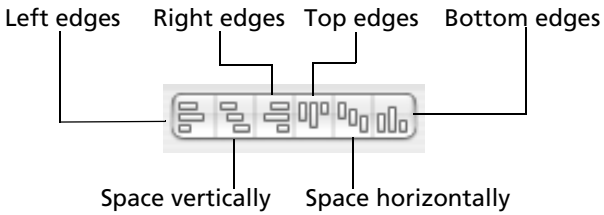
If you have selected the Show Control Order option while you modify the control order using the Properties pane, changes to the control order for all the controls are reflected when you save a new ControlOrder value using the Properties pane.

Aligning Controls with Other Controls

REALbasic's Interface Assistant makes it easy to align a particular control with another control. Simply drag the control until it is close to being aligned with the other control. When you get close to aligning the two controls, horizontal and/or vertical alignment rules will appear. When you release the mouse button, REALbasic will snap the control you are dragging into place.

To align controls that have already been placed in the window, you use the Alignment icons on the right side of the Window Editor's toolbar. You can also use the equivalent menu command in the Edit ▶ Align submenu.

Figure 153. The Window Editor's Alignment and Spacing icons.



The icons align the selected objects as shown in Figure 153. The Edit ▶ Align submenu has the same commands.

The Space Vertically and Space Horizontally commands distribute three or more objects evenly. They are described in the section that follows immediately.

If you need to align several controls, do this:

1. **Click on the control whose position is already correct to select it.**
2. **Click the Back button in the Window Editor toolbar or choose Edit ▸ Arrange ▸ Send to Back to insure that the selected control remains in place while the other controls move to align with it.**

   You can also move the control to the back by setting its ControlOrder property to zero using the Properties pane.
3. **While holding down the Ctrl key (Command key on Macintosh), select each of the controls you wish to align or draw a selection rectangle around the controls to be aligned.**
4. **Click the desired alignment icon in the Window Editor toolbar or choose the equivalent menu command from the Edit ▸ Align submenu.**

Spacing Controls Evenly

REALbasic provides an easy way to reposition controls to evenly distribute empty space between them. The row of alignment icons (Figure 153) also contain icons for distributing controls evenly along the horizontal and vertical axes.

To distribute the controls evenly, do this:

1. **Click on a control to select it.**
2. **Hold down the Ctrl key (Command key on Macintosh) and select at least two other controls or draw a selection rectangle around the additional controls.**
3. **To distribute the controls horizontally, click the Space Horizontally icon or choose Edit ▸ Align ▶ Space Horizontally.**
4. **To distribute the controls vertically, click the Space Vertically icon or choose Edit ▸ Align ▶ Space Vertically.**

   Your selected controls will now be equally spaced in either the horizontal or vertical axis.

The Control Hierarchy

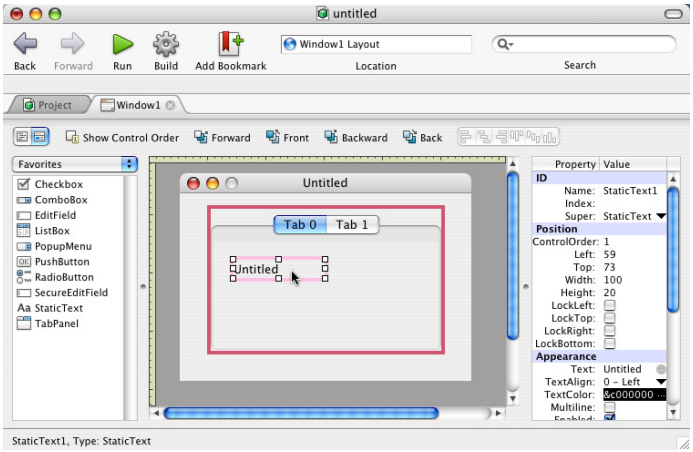
In some cases you build portions of your interface by placing some controls within another control. For example, you use the GroupBox control to organize other controls, usually RadioButtons, CheckBoxes, PopupMenus, and EditFields. The TabPanel and PagePanel controls also designed to enclose other controls. The Control Hierarchy in the IDE enables you to work with all the controls as a group.

The control that encloses the others is known as the *parent* control and the controls that are entirely within its borders are the *child* controls. This works automatically if you create the controls in the order of: parent-child. That is, create the control that encloses the others first, then create the child controls or duplicate existing child controls and keep them inside the parent control.

If you create the control that is supposed to be the parent after creating some of the other controls, you can convert it to the parent by moving it to the back in the control order. Do this by selecting the control that is supposed to be the parent and click the Back button in the Window Editor toolbar or choose Edit ▶ Arrange ▶ Send to Back.

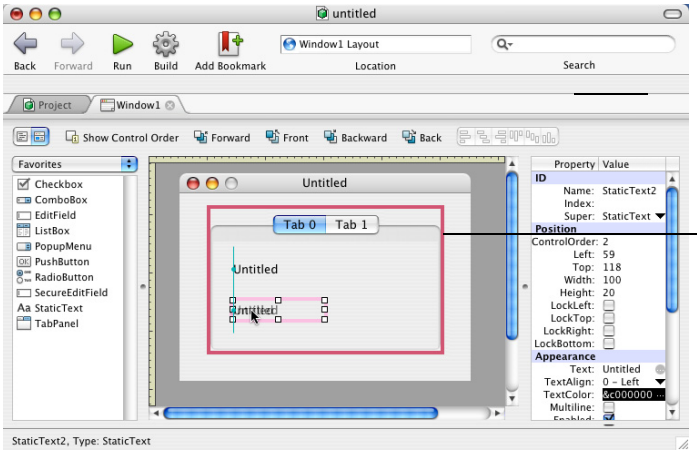
When you drag a child control to a parent and then drop it into place, a marquee surrounds the parent. This gives you visual confirmation that you are, in fact, adding a child to a parent. For example, in Figure 154 a StaticText control was just dropped onto a TabPanel.

Figure 154. Adding a child control to a parent.



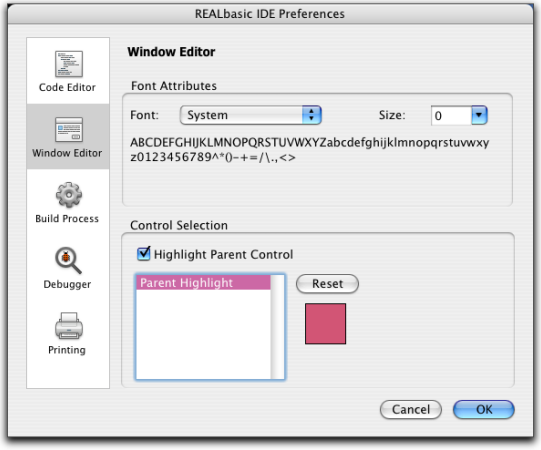
By default, a marquee also surrounds the parent control whenever you select an existing child control. For example, in Figure 155 the user clicked the second StaticText control and is aligning it with the top StaticText. A marquee surrounds its parent (TabPanel) control as well.

Figure 155. Selecting and aligning a child control.



An option in the Window Editor options panel turns this feature on or off. By default, this feature is enabled; to disable the feature, choose Edit ▶ Options (REALbasic ▶ Preferences on Macintosh) and select the Window Editor panel and deselect Highlight Parent Control.

Figure 156. The Window Editor preferences.



You can additionally select the color of the marquee. Click the color sample to display the Color Wheel to select another color. Click Reset to revert to the original color.

If a control is not completely enclosed by another control, then it doesn't automatically become a child; it simply overlaps the other control.

If you move a child outside its parent, it is no longer a child of that control. If you move it completely inside another control, it becomes the child of that control.

When you copy a parent control, you copy all its child controls as well.

You can create more than one level of nesting. For example, you can place a GroupBox within a TabPanel and then place several RadioButtons within the GroupBox. In this case, the GroupBox is the parent of the RadioButtons and the TabPanel is the parent of the GroupBox. To make this work automatically, you must create them in the order that respects the hierarchy: First create the TabPanel, then the GroupBox, and then the RadioButtons.

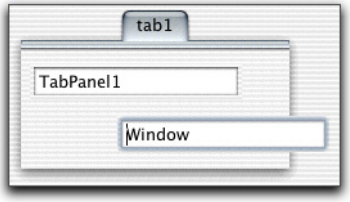
However, you cannot nest a TabPanel in another TabPanel or a PagePanel and you can't nest a PagePanel in either a TabPanel or another PagePanel.

Control Hierarchy Features

If you duplicate a child control and leave the duplicate within the parent, then it is automatically a child. However, if you move the duplicate outside the parent, it is no longer a child. A control must be fully enclosed by the parent to be considered a child.

In Figure 157 the bottom control was duplicated from the top one, but has been moved out of the TabPanel. It is no longer a child.

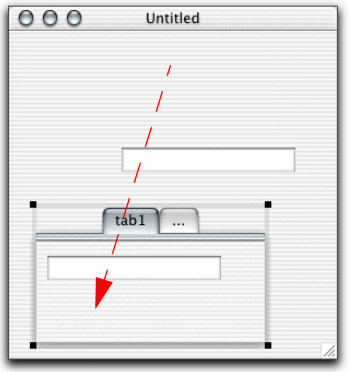
Figure 157. A child and an 'unrelated' control.



If you move the bottom EditField back within the TabPanel, it becomes a child of the TabPanel once again.

Moving a parent control moves its child controls as well. In Figure 158, the TabPanel shown in Figure 157 was moved down. It takes the top EditField with it, but leaves the bottom one orphaned.

Figure 158. Moving a parent control.



- Deleting a parent control deletes all child controls.
- Hiding a parent hides all child controls, but retains the previous visibility status of all children.
- Showing a parent control shows only the child controls whose visibility is set.
- Disabling a parent control disables all its child controls, but retains the previous visibility status of all children. In the IDE, disabling a container visually disables all the child controls, but does not update the Enabled property.

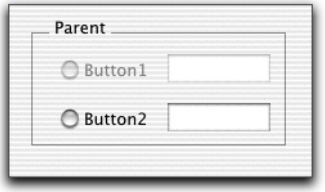
  In Figure 160, a disabled GroupBox in which two RadioButtons and two EditFields are children is shown in the Development environment.

Figure 159. A GroupBox disabling its children.

When the GroupBox is disabled, all four children are disabled. Since all children are disabled when the parent is disabled, the EditFields are not enterable in the application.

Enabling a parent control enables only the child controls whose Enabled property is set. In Figure 160, only the bottom row of child controls have the Enabled property set. When the GroupBox is enabled, the top row of controls remains disabled.

Figure 160. A enabled GroupBox control.



If this behavior is not desirable, you can break the control hierarchy by setting the Parent property of any child control to Nil in the Open event of the window. You can also break the control hierarchy by moving the child control in back of the parent control in the control order. The enclosed control with then behave independently of it former "parent" in the built application.

Adding Menus and Menu Items

REALbasic has a built-in Menu editor that makes adding menubars, menus, and menu items to your project easy. You can use the default menubar for the entire application or create other menubars that are used for certain windows.

Adding Menubars

When you first create a Desktop Application project, the project includes one window and one menubar, named MenuBar1. This is the application's default menubar. By default, it is used for your entire application. However, you can add additional menubars to your project and associate a menubar with a window. On Linux and Windows, each window has its own menubar, so each window's menubar is always used for that window. On Macintosh, when a window becomes active, the window's menubar replaces the current menubar.

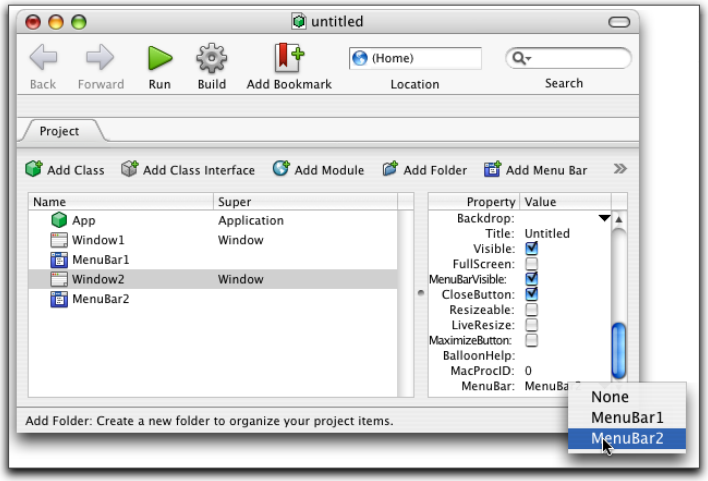
To add an additional menubar to your project, do this:

1. **If it is not already displayed, click the Project tab to view the Project Editor.**
2. **Click the Add Menu Bar button or choose Project ▶ Add ▶ Menu Bar.**

   A new menubar appears in the Project Editor, named MenuBar2.
3. **Follow the steps in the following section, "Adding Menus" on page 156 to add menus and menu items to the menubar.**
4. **To assign a menubar to a window, use the window's Properties pane to set the window's MenuBar property to the name of the menubar.**

   The window's MenuBar property is listed in the Appearance group in the Properties pane. All the menubars that belong to the project appear in this pop-up menu.

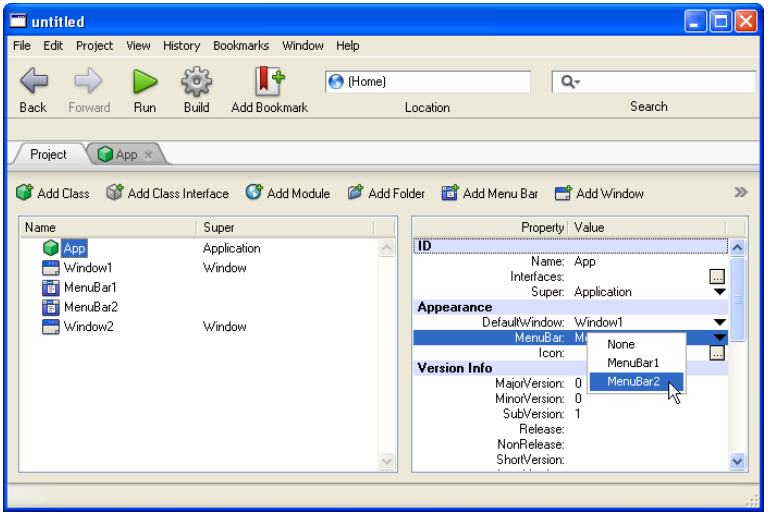
Figure 161. Changing a window's menubar.



This feature is especially relevant for Windows and Linux applications because each window can have its own menubar. If you use this feature on Macintosh, the global menubar at the top of the screen switches when a window with a different menubar is brought to the front.

5 (Optional) To change the default global menubar, set the App class's MenuBar property to the desired menubar.

Figure 162. Changing the default global menubar after adding a second menubar.



On Windows MDI applications, the global menubar is used as the enclosing frame's menubar.

Adding Menus

There are four steps for implementing a menu and its items. These steps are the same for the default global menubar and any additional menubars that you add to the project.

- Adding the menu to a menubar using the Menu Editor,
- Adding the menu's items using the Menu Editor,
- Enabling the menu item. A disabled menu item is grayed out and is not selectable.
- Adding a menu handler. A menu handler is the method that tells REALbasic what to do when the user chooses the (enabled) menu item. An enabled menu item without a menu handler can be selected but it does nothing.

  There are two ways to enable a menu item. If the menu item is to be enabled all the time, you can select the menu item's AutoEnable property in the menu item's Properties pane. When you add the menu handler, the menu item is functional (If you don't write the menu handler, the question of enabling the item is moot because the menu item would do nothing without a menu handler). For example, you might want to use the AutoEnable property for the New menu item in the File menu which creates a new document in the application.

  If the menu item should not be enabled all the time, deselect the AutoEnable property. In this case, the menu item is disabled by default. You need to tell REALbasic when to enable it. For example, you might have an Export menu item that should be enabled only if the user has selected an item to export.

  When AutoEnable is False, the menu item is disabled until the user attempts to pull down a menu. At that moment, you can decide if conditions are right for the menu item to be enabled. For example, a Save menu item would need to check that the document has never been saved or that changes have been made to the document since the last save operation. For information about enabling menu items, see the section, "Enabling Menu Items" on page 290.

  For information about creating a menu handler, see the section, "Adding Code To a Menu Item" on page 290.

Note You can also add or remove menu items via code. For more information, see the entry for the MenuItem class in the *Language Reference*.

REALbasic adds File and Edit menus to your project automatically. On Mac OS X, it also adds the Apple and Application menus. On Mac OS "classic," it adds the Apple menu.

Desktop Applications normally have a File menu with a Exit (Quit on Macintosh) menu item. You can remove the Edit menu only if your application has no controls that could be edited by the Edit menu items.

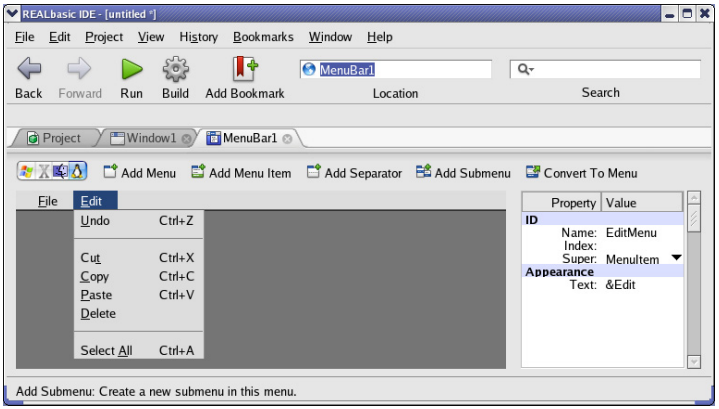
To add a menu to a menubar, do this:

1 If it is not already displayed, click the Project tab.

2 Double-click on a Menubar to open its Menu Editor.

The Menu Editor appears.

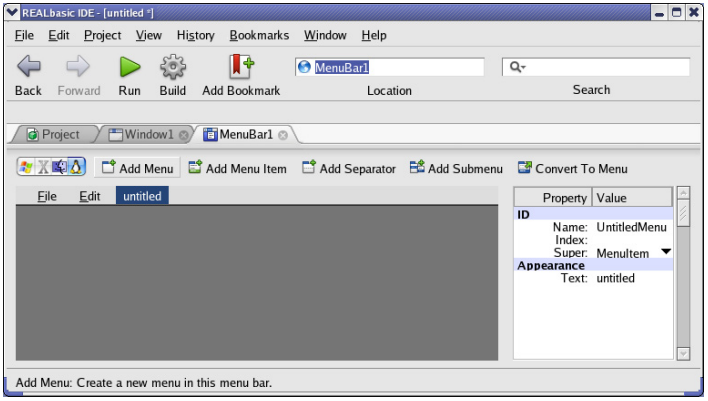
Figure 163. The Menu Editor.



3 Click the Add Menu button or choose Project ▸ Add ▸ Menu.

A new menu is added to the menubar as the last menu, or, if a menu is selected, to the right of the selected menu. Its properties are shown in the Properties pane.

Figure 164. The Menu Editor after adding a menu.



4 (Optional) If the new menu does not appear in the desired position in the menubar, drag it to its new position.

By default, the new menu is added to the right of the existing menu bar menus.

5 **In the Properties pane for the new menu, enter the Name of the menu and the Text that will appear in the menubar.**

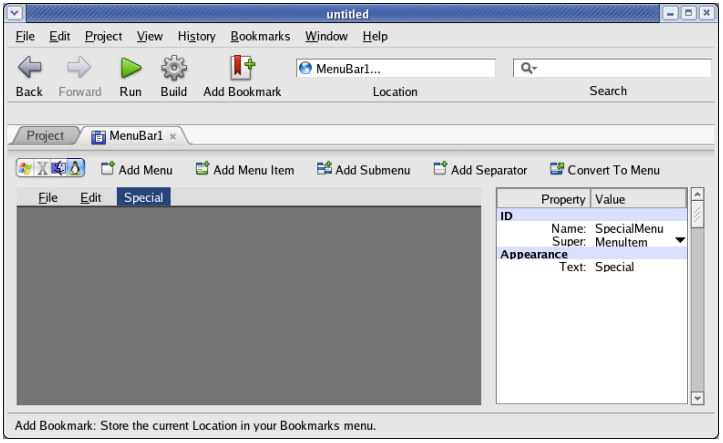
Table 2: Properties of a MenuItem.

Name	Description
Name	The internal name of the Menu used to identify it in programming code.
Super	The class of object the Menu control is based on. This will normally be MenuItem. You can also use PrefsMenuItem, AppleMenuItem, or QuitMenuItem.
Text	The text that will appear in the Menu bar. To designate a keyboard accelerator for the menu place the "&" prior to the accelerator key. For more information, see the section "Accelerator keys" on page 160.

If you enter the Text property first and then press Return, REALbasic will suggest a name.

Figure 165 shows a new menu and its properties in the Properties pane.

Figure 165. A new menu in the Menu Editor.



Adding a Help Menu

Many applications have a Help menu that is the right-most menu in the application. The Help menu may contain menu items that give the user access to an application-specific help system. You can add a Help menu to your project.

To add a Help menu, do this:

1 **Add a menu to the end of your menubar.**

2 **Set the Text property of the menu to Help.**

Adding Menu Items

The Menu Editor makes it easy to add menu items to your menus. You can assign keyboard shortcuts to menu items, but remember that the operating system looks for a shortcut starting from the leftmost menu. That means that if you assign the same keyboard shortcut to two different menu items, one of them won't work.

There are also several specific keyboard shortcuts that are reserved for specific functions. These are:

Table 3: Reserved Keyboard Shortcuts

Menu	Keys (Windows and Linux)	Keys (Macintosh)	Command
File	Ctrl-N	⌘-N	New
File	Ctrl-O	⌘-O	Open…
File	Ctrl-W	⌘-W	Close
File	Ctrl-S	⌘-S	Save
File	Ctrl-P	⌘-P	Print…
File	Ctrl-Q	⌘-Q	Quit
Edit	Ctrl-Z	⌘-Z	Undo
Edit	Ctrl-X	⌘-X	Cut
Edit	Ctrl-C	⌘-C	Copy
Edit	Ctrl-V	⌘-V	Paste
Edit	Ctrl-A	⌘-A	Select All
Edit	Esc	⌘-period	Terminate an operation
Edit	Ctrl-M	⌘-M	Minimize

Windows XP has the following system-wide keyboard equivalents that you should be aware of:

Table 4: Windows keyboard equivalents.

Key Combination	Description
Alt+Esc	Switches to the next application.
Alt+F4	Closes an application or window.
Alt+hyphen	Opens the window menu for a document window.
Alt+Print Screen	Copies an image in the active window onto the Clipboard.
Alt+Spacebar	Opens the window menu for the application's main window.
Alt+Tab	Switches to the next application.
Ctrl+Esc	Switches to the Start menu.
Ctrl+F4	Closes the active group or document window.
F1	Starts the application's Help file, if it exists.
Print Screen	Copies an image of the screen onto the Clipboard
Shift+Alt+Tab	Switches to the previous application. The user holds down both Shift and Alt while pressing Tab.

The MenuItem class has the following properties that you use to specify the keyboard shortcut:

Table 5: Properties for specifying a keyboard shortcut.

Property	Date Type	Description
Key	String	The key that the user presses to trigger the menu item's event handler. Enter a single uppercase letter. It is pressed in combination with one or more modifier keys, specified with the other properties.
MenuModifier	Boolean	If True, the user must hold down the Ctrl key (Windows and Linux) or Command key (Macintosh) when pressing the Key key. When a value is entered for the Key property, this property is set to True by default.
AlternateMenu Modifier	Boolean	If True, the user must hold down the Shift key when pressing the Key key.
PCAltKey	Boolean	If True, the user must hold down the Alt key when pressing the Key key. This property is for Windows and Linux only.
MacOptionKey	Boolean	If True, the user must hold down the Option key when pressing the Key key. This is a Macintosh-only property.
MacControlKey	Boolean	If True, the user must hold down the Control key when pressing the Key key. This is a Macintosh-only property.

For example, to specify the keyboard shortcut Ctrl+H, enter "H" as the Key property and select the MenuModifier property. To specify Shift+Ctrl+H, you would select both the MenuModifier and AlternateMenuModifier properties. The keyboard shortcut Alt+Ctrl+H requires the PCAlt and MenuModifier key properties.

When you make your selections, the keyboard equivalent is shown to the right of the MenuItem's text in the Menu Editor.

Accelerator keys

The Windows and Linux platforms also have the concept of keyboard accelerators. In addition to the keyboard equivalent, which work on all platforms, you can also add a keyboard accelerator for each menu and menu item. When you designate a key as the keyboard accelerator, it is underlined in the menu name or menu item. The user can display the menu or invoke the menu item by holding down Alt and pressing the accelerator key. With a comprehensive system of accelerators, a user can use the menu system without using the mouse at all.

To designate the accelerator key, precede the letter by an ampersand ("&") in the menu or Menuitems's Text property. For example, if you are creating a menu named "Actions" and you want to make the keyboard accelerator the "a', you would enter "&Actions" as the menu's Text property. To make the "t" the accelerator key, enter "Ac&tions". Keep in mind that accelerators are not shown and do not work on Macintosh.

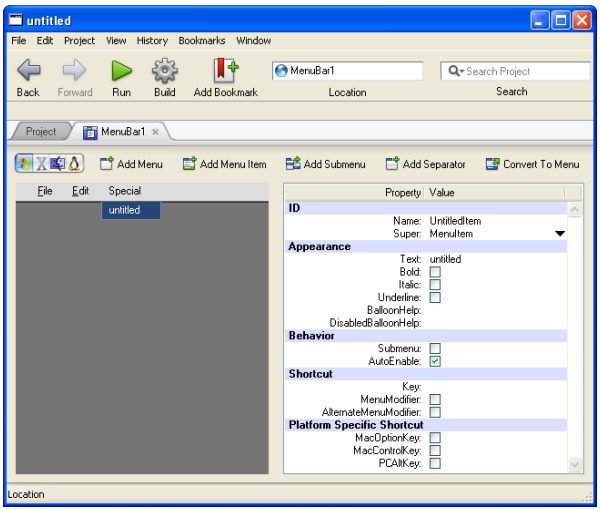
To add a menu item to a menu, do this:

1 **In the Menu Editor, select the menu you wish to add a menu item to by clicking on it.**

2 **Click the Add Menu Item button or choose Project ▶ Add ▶ Menu Item.**

A new menu item is added to the selected menu and its properties are shown in the Properties pane.

Figure 166. A new menu item added to the menu.



3 **In the Properties pane, enter the Text property for the menu item and press Enter.**

To create a keyboard accelerator for Windows and Linux, precede the accelerator key with the "&" symbol.

The Menu Editor will supply a default Name. If you wish, you can replace the suggested Name with a name of your own.

4 **If desired, add a keyboard shortcut by assigning a letter to the Key property and at least one of the modifier key properties listed in Table 5 on page 160.**

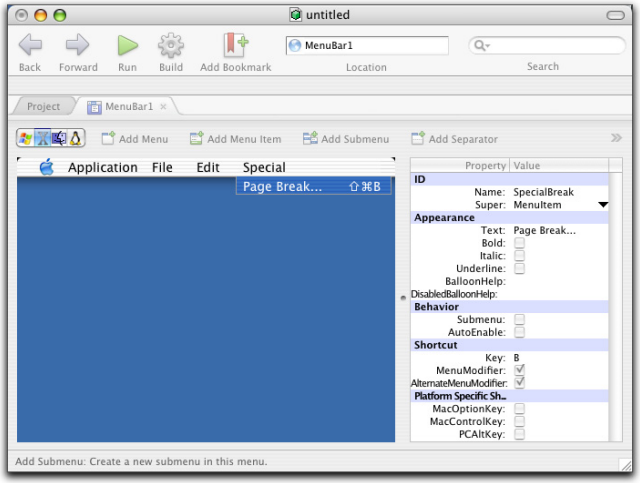
5 **If desired, disable the AutoEnable property if you need to enable the menu item only under certain conditions.**

6 **(Optional) Select the Bold, Italic, or Underline properties.**

The BalloonHelp and DisabledBalloonHelp are for Mac OS "classic" builds only. Consider adding Balloon Help only if you will deploy a Mac OS "classic" version of the application.

Figure 167 on page 162 shows a Page Break menu item added to the Special menu with the Keyboard equivalent Shift-Command-B (Shift-Ctrl-B on Windows and Linux).

Figure 167. A new menu item in the Special menu.



NOTE: Although the Menu Editor allows you to use lowercase characters as keyboard shortcuts, only uppercase characters should be used.

Adding a Submenu

Submenus are menu items that display an additional menu to their right. The menu item itself is not selectable. It acts only as a title for the submenu.

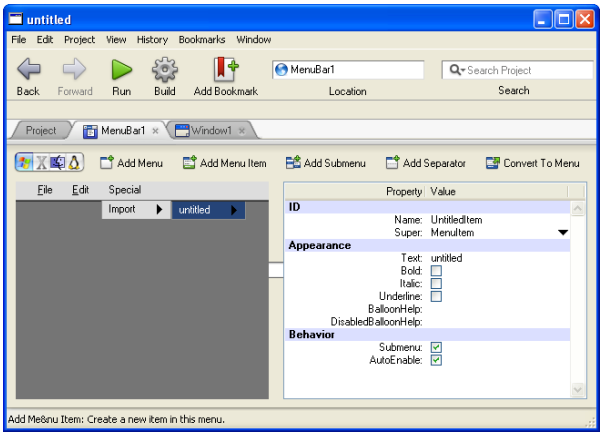
To add a submenu to an existing menu item, do this:

1. **Click on the menu item in the Menu Editor to select it.**
2. **In the Properties pane, place a checkmark in the Submenu property.**

   The Menu Editor adds an arrow to the right of the menu item to indicate that it is the parent of a submenu.
3. **Click the Add Submenu button or choose Project ▸ Add ▸ Submenu.**

   In the Menu Editor, a submenu item is added to the menu item you selected in step 1.

Figure 168. A submenu item added to the Special ▶ Import menu item.



4 **In the Properties pane, enter the Text for the submenu Item and press Enter.**

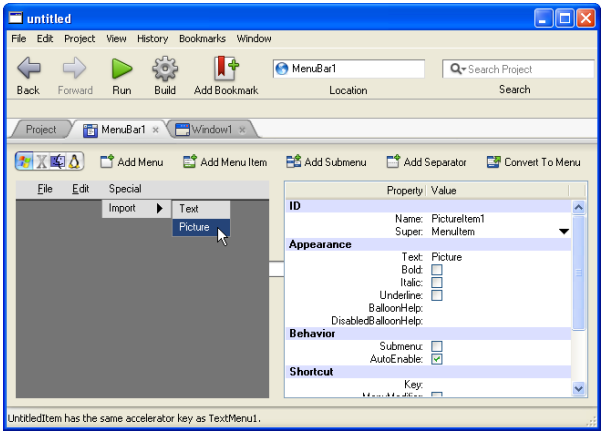
5 **To add an additional submenu item to this submenu, select this submenu item and click the Add Menu Item button in the Menu Editor toolbar.**

In the Menu Editor, REALbasic adds the new submenu item to the current submenu.

6 **Use the Properties pane to enter the Text of this submenu item and press Enter.**

REALbasic suggests a default Name for the new submenu item. If you wish, you can change it.

Figure 169. A submenu with two submenu items.



If you wish, you can continue to add a submenu to an existing submenu, creating a three-level hierarchical menu system. However, submenus can be difficult to navigate for the new computer user. They also hide menu items from view. If a user

scans through the menu items looking for a particular menu item, he may not look at the submenus. Consider the audience for your application before using submenus. If many of your users will be new computer users, consider displaying a dialog box to choose the functions you could put in a submenu.

Adding a Menu Item to the Mac OS Apple and Application Menus

Mac OS X applications add a new menu between the Apple menu and the standard File menu. It automatically takes on the name of the application. When you create a standalone Mac OS X application, the menu's name is the name of the Macintosh version of the application that you entered in the App class's Properties pane. (see Figure 385 on page 550).

Although the Application menu appears in the Menu Editor when you use the Mac OS X preview, you cannot add menu items directly to it or to the Apple menu. Instead, you use two special MenuItem classes to install menu items onto those menus, PrefsMenuItem and AppleMenuItem. Use the PrefsMenuItem class to manage your Preferences menu item and the AppleMenuItem class to manage items that should appear in the Apple menu on Mac OS "classic." You cannot add menu items to the Apple menu on Mac OS X.

For your preferences menu item, put the menu item where you want it to appear under Windows and Linux and set their Super class to PrefsMenuItem. A MenuItem based on the PrefsMenuItem class will be moved automatically to the Application menu for the Mac OS X build.

According to Apple's Mac OS X user interface guidelines, you should use ⌘-, (Command-comma) as the keyboard shortcut for an application's Preferences menu item.

Mac OS "classic" adds only the Apple menu to the left of the File menu. If you want to add a menu item to this menu, you should place the menu item in the menu that will be used for your Windows and Linux builds and base this menu item on the AppleMenuItem class. On your "classic" build, it will move to the Apple menu and on the Mac OS X build, it will move to the application menu.

In theory, you could also base a MenuItem on the QuitMenuItem class. Menuitems that are based on QuitMenuItem automatically call the Quit method and quit the application. Since a Quit MenuItem is included in the File menu by default, you are not required to add such an item to your project.

The Exit (or Quit) menu item

The QuitMenuItem class is intended only for the Quit (or Exit) menu items and causes the menu item to move to the application's menu for Mac OS X builds. Its Text property uses a constant rather than a text literal. The constant is defined in the App class's Code Editor. With a constant, the menu item uses the text "Quit" on Mac OS builds (both Mac OS X and "classic") and "Exit" on Windows and Linux builds. The use of constants for this purpose is discussed in the section "Using Constants to Localize your Application" on page 307.

Moving Menus and Menu Items

A menu item can be moved to a new position in the menu by dragging the menu item. You can only move a menu item to another position on the same menu. If you need to move the menu item to another menu, you have to delete it and recreate it on the other menu.

To move a menu item, do this:

1. **Click on the menu item you want to move to select it.**
2. **Drag the menu item towards the position on the menu where you want it.**
3. **When the menu item is in the desired position, release the mouse button.**

You can also move menus within the menu bar. In a similar fashion, drag a menu in the menu bar and move it to the left or right. Drop the menu when it is between the desired menus.

Converting a Menu Item to a Menu

You can also convert a menu item to a menu. To do so, select the menu item and then click the Convert To Menu button in the Menu Editor toolbar or choose Project ▶ Modify ▶ Convert to Menu. The menuitem is then removed from its menu and appears in the menubar. From there you can drag it to another position in the menu bar if you wish.

Removing Menu Items

To remove a menu item from a menu, do this:

1. **In the Menu Editor, click on the menu item to select it.**
2. **Press the Delete key or choose Edit ▶ Delete.**

Adding A Menu Item Separator

Menu item separators are lines that appear in between menu items to logically group items together. To add a menu item separator, simply select a menu item and click the Add Separator button in the Menu Editor toolbar or choose the Project ▶ Add ▶ Separator command. The separator will appear just below the selected menu item. If you wish, you can drag it vertically to a new location.

Menu Item Arrays

In certain cases, you cannot specify the menu items that belong in a menu in advance. They may change depending on the context in which the application is used or on the computer environment in which the application is running.

A common example of this is the Font menu that is normally included in any application that supports styled text. The programmer has no way of knowing in advance which fonts happen to be installed on the user's computer.

You begin by creating the Menu normally. You then create a menu item, name it, and set its Index property to 0 to start the menu item array. It doesn't matter what its Text property is. You then must write code to populate the array with the names of the fonts installed on the user's computer. If we assume that fonts won't be added or deleted while the application is running, we can build the font list when the application starts up.

To do this, place the code in the App class's Open event handler. This runs when the application starts up. The App class that is added to your project by default is the place for this.

The following code populates the Font menu when the application opens. It uses the built-in Font function which returns the name of the i^{th} font on the user's computer. It assumes that the Font menu item that was added manually is named FontFontName and has the Index of zero.

```
Dim m as MenuItem
Dim i, nFonts as Integer

nFonts=FontCount-1
//build the font menu
FontFontName(0).text=Font(0)
For i=1 to nFonts
 m=New FontFontName //create new menu item
 m.Text=Font(i) //obtain name of ith font
Next
```

If the AutoEnable property is set to True and the dynamically created menu items have menu handlers, they will be enabled automatically.

Importing and Exporting Menus

Menubars can be exported to the desktop and imported into other projects. When you export a menubar, only the menubar is exported, not the menu items' menu handlers.

To export a menubar, do this:

1. **In the IDE, click the tab of the menubar you want to export or, if the menubar is not open, click on it in the Project Editor.**
2. **Choose File ▸ Export *MenubarName*, where *MenubarName* is the name of the menubar you have selected in the previous step.**

   A save-file dialog box appears.
3. **Navigate to the desired directory, enter a filename, and click Save.**

   REALbasic saves the menubar as a .rbm file. Its desktop icon looks like this:

Figure 170. An exported menubar.

To import a menubar, do this:

1. **From any IDE editor, choose File ▸ Import.**

   A standard open-file dialog box appears.

2 **Navigate to the directory that contains the exported menubar, select it, and click Import.**

REALbasic adds the imported menubar to the Project Editor. It uses the name it had in the project from which it was exported.

User Interface Guidelines

The quality of your application's interface will determine how successful your user will be in using it. It's absolutely critical that your users find the interface intuitive. Studies have shown that if a user can't accomplish something within the first 15 minutes of using an application, he will give up in frustration. Beyond simply being intuitive, the more polished an application's interface is, the more professional it will appear to the user. Remember that without realizing it, your users will be comparing your application's interface to all of the other applications they have used.

Each platform that REALbasic supports has its own conventions. User interface guidelines are available from the following sources:

- **Windows:** Microsoft's User Interface guidelines at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/welcome.asp
- **Macintosh:** Apple Human Interface guidelines: http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/
- **Linux KDE Desktop:** KDE user interface guidelines: http://developer.kde.org/documentation/design/ui/

  KDE is one of the most popular Linux desktops and is used by default in several major Linux distributions. However, there are others. Linux supports a greater degree of desktop customization than Windows and Macintosh.

- REALbasic's Interface Assistant™ helps you create a nice interface by making it easy to align controls with other controls. For example, when you drag a control near the edge of the window, REALbasic displays an alignment line to let you know where to stop. For the left, right and bottom edges, it stops at 20 pixels and for the top edge, it stops at 14 pixels. When dragging a control towards another control, REALbasic displays an alignment line at 12 pixels (the recommended distance between controls).

  But there is more to a professional, polished interface than simply aligning controls. We all think we know how to create a nice interface because we have used lots of applications. But using an interface is a lot different from designing on.

CHAPTER 4

BASIC Programming Concepts

Programming is all about getting the computer to do what you want it to do. The key is knowing how give the computer instructions in a way it will understand. That's where programming languages come in. There are many different programming languages that are designed to make the communication easier in different situations.

In this chapter you will learn about the BASIC programming language, how it is different in REALbasic, and the fundamentals of programming.

Contents

BASIC versus REALbasic

The BASIC language was created in the 1960's for the purpose of teaching people programming. Most of what made other languages difficult to master was removed from BASIC to make learning it easier. In fact, BASIC is an acronym that stands for Beginners All-Purpose Symbolic Instruction Code.

For a long time BASIC was considered less powerful than other languages, but this was mostly due to the way it was implemented rather than the language itself. Spoken languages wouldn't be considered to be very powerful if you could only speak one word every 10 minutes. Computers actually only understand two things, 1 and 0. That's it. That's all they know. The rest of what a computer does all breaks down to that fundamental concept. Every command you give your computer is eventually translated into a series of instructions that consist only of 0's and 1's. These 1's and 0's that computers understand are referred to as *machine language*.

Most versions of BASIC used an interpreter program to execute the code. This means that each time a program ran, the BASIC interpreter had to turn the BASIC code into machine language. Other languages had compilers which are special programs that translate the programing language into machine language all at once. This makes programs execute faster because the real-time interpretation is removed.

REALbasic has a compiler built-in to it. That means your code runs as fast as possible. BASIC is a traditional programming language that starts with the first line of programming code and continues until the last line. REALbasic is a modern, object-oriented version of BASIC. If you are new to programming that might not mean much now but it will. REALbasic takes the simplicity of the BASIC language and adds the power of modern programming through its object-oriented implementation and compiler. Also, most programming languages require you to know quite a bit about how to communicate with the computer's operating system. REALbasic abstracts you from all of that making it easier for you to learn and easier to run your application on computers running operating systems that are different from the one you created your application on.

Storing Values in Properties and Variables

When you need to store information so you can access it again even after you have shut off your computer, you tell your computer to store the information in a document. When a computer needs to store information temporarily, it is stored in the computer's memory. The computer's memory is like a series of organized boxes. Each box has a location in memory with an address that is used to locate it. These locations are given names to make them easier to work with. Depending on how these memory locations are used, they are called variables and properties.

What are Properties?

The variables that store values that make up the description of an object — such as a window — are called *properties*. The title of a window is stored in a property. The width of the window is a property. When a window is opened, the values of these properties are copied into memory. You can access them using their names. You can get values from them and you can store new values in them. For example, if you wanted the title of a window to change when the user clicks a button, you would set the title property of the window to the new value.

Each property can hold a certain type of data. Some properties store text (like a window title) while others store numbers (like the window's width property). Later in this chapter, you will learn how to assign values to properties and how to get the values that are stored in properties.

Variables

Sometimes you will need to store a value that isn't related directly to an object like a window or a button. In this case you use a variable. A variable is just like a property but it isn't directly related to any particular object. Later in this chapter, you will learn how to create variables, assign values to them and get values from them.

Data Types

To make programming code execute faster and to provide powerful commands that save you time when programming, computers have to be able to make certain assumptions about the information you give them. For example, when you give a computer a piece of information, the computer needs to know if it's a number, a string of characters, a date, etc. If you didn't tell the computer what kind of data you are giving it, it wouldn't know whether you meant 1 plus 1 to be 2 or the string "11". In this example, telling the computer that you are giving it numbers will result in 2. Telling it you are giving it simply a string of typed characters will result in the string "11". There are many data types that REALbasic understands but there are six data types that are by far the most common. They are String, Integer, Single, Double, Boolean, and Color.

One important programming convention in REALbasic is that you must tell REALbasic the data type of each variable and property that you create. It does not 'infer' the data type from the way you use your variables and properties.

String

A String is just a series (or string) of characters. Basically any kind of information can be stored as a string. "Jeannie", "3/17/98", "45.90" are all examples of strings.

You might be thinking "Hey, those last two don't look like strings" but they are. When you place quotes around information in your code, you are telling REALbasic to look at the data as just a string of characters and nothing more. The maximum length of a string is based only on available memory.

You can combine two strings with the addition symbol (+). For example, the statement "Big" + "Dog" results in the string "BigDog". That is really the extent of the "mathematics" you can perform on strings. However, REALbasic has many built-in functions that make processing strings easy. For example, the Lowercase function takes a string and converts all the characters to lowercase. The Trim function trims off any leading and trailing whitespace characters.

Integer

An integer is a whole number. It cannot accept a decimal or fractional value. REALbasic offers both signed and unsigned integer data types that use one, two, four, or eight bytes of memory. The following table summaries these data types.

Data Type	Number of Bytes	Range
Int8 or Byte	1	-128 to 127
Int16	2	-32,768 to 32,767
Int32 or Integer	4	-2,147,483,648 to 2,147,483,647
Int64	8	-2^{63} to $2^{63}-1$
UInt8	1	0 to 255
UInt16	2	0 to 65535
UInt32	4	0 to 4,294,967,295
UInt64	8	0 to $2^{64}-1$

REALbasic's Integer data type is a signed integer that uses the word length for the target platform. Currently this is four bytes on all supported platforms.

Because integers are numbers, you can perform mathematical calculations on them. Unlike strings, integers do not have quotes around them in your code.

Single

A Single is a number that can contain a decimal value. The range of a Single is shown in Table 6. In other languages, REALbasic's Single may be referred to as a single precision real number. Because Singles are numbers, you can perform mathematical calculations on them. Single numbers use 4 bytes of memory.

Double

A Double is a number that can contain a decimal value. The range of a Double is shown in Table 6. In other languages, REALbasic's Double may be referred to as a double precision real number. Because Doubles are numbers, you can perform mathematical calculations on them. Doubles use 8 bytes of memory. Since Doubles use more decimal places to represent a number, you would want to use Doubles rather

than Singles if the extra precision is important in your calculations. However, calculations on Singles are generally faster than Doubles.

Table 6 gives the upper and lower limits of integers, singles, and doubles:

Table 6: Minimum and Maximum values for Integers, Singles, and Doubles.

Data Type	Smallest Value	Largest Value
Integer	-2147483648	2147483647
Single	1.175494 e-38	3.402823 e+38
Double	2.2250738585072013 e-308	1.7976931348623157 e+308

Boolean

A Boolean can take on the values of either True or False. Boolean values are False by default but can be set to True using REALbasic's True function and back to False using the False function. Some of the properties of objects in REALbasic are boolean values. For example, most of the controls have an Enabled property that is boolean.

Color

A Color is an intrinsic data type that stores the value of a REALbasic color. A Color "value" actually consists of three numeric values that can be set using any of the three popular color models, Red-Green-Blue, Hue-Saturation-Value, or Cyan-Magenta-Yellow. Each value is stored as a byte. That means that each number can take on 256 possible values.

A Color can also be set to a value via one of the three color functions that correspond to the three color models. Table 7 summarizes your options.

Table 7: REALbasic functions for specifying a color.

Function	Color Model	Data type and range of Parameters
RGB	Red-Green-Blue	Integer (0-255)
HSV	Hue-Saturation-Value	Double (0-1)
CMY	Cyan-Magenta-Yellow	Double (0-1)

For example, the following code assigns a color to the FillColor property of a rectangle:

```
Rectangle1.FillColor=CMY(.35,.9,.6)
```

Rectangle1 is the name of a rectangle object in a window and FillColor is one of its properties. The data type of FillColor is color, so it only takes values that correspond to a color specification. If you tried to assign a value of a different data type, you would get an error message.

You can also assign a color using the Red-Green-Blue model using the format:

```
&c<i>RRGGBB</i>
```

where *RR* is the value of Red in hexadecimal, *GG* is the value of Green in hexadecimal, and *BB* is the value of Blue in hexadecimal. Each value ranges from 00 to FF (FF is 255 in hexadecimal). The expression "&c" signals that the next six characters make up a color value in hexadecimal.

For example, you can write an expression such as:

```
Rectangle1.FillColor=&cFF594A
```

You can use any of the three color functions interchangably. After you assign a color, you can read the value of any of its nine properties.

Variant

A Variant is a special data type that can store the value of any data type, including objects. The Variant's Type method returns an integer that identifies the data type that the variant is storing:

Value	Description
0	Nil
2	Integer
5	Double or Single
7	Date
8	String
9	Object
11	Boolean
16	Color

Depending on the data type of the value stored by the Variant, you can use one of the following properties to convert the value to another data type:

Property	Description
BooleanValue	Returns the value of the variant as a Boolean.
ColorValue	Returns the value of the variant as a Color.
DateValue	Returns the value of the variant as a Date. When retrieving the string value of a date, the string is returned in SQL date format *YYYY-MM-DD HH:MM*.
DoubleValue	Returns the value of the variant as a Double.
IntegerValue	Returns the value of the variant as an Integer.
ObjectValue	Returns the value of the variant as an Object.
StringValue	Returns the value of the variant as a String.

Although you use a Date object to store date and date/time values, it is not a data type. Technically, Date is a class. See the section "Declaring Objects" on page 182 for information on using the Date class.

Structures

A structure is a compound data type. It consists of a list of fields, where each field has its own data type. It is similar to Visual Basic's "user-defined types" and Visual Basic .net's structures.

Structures can be created only in REALbasic modules, while all the other data types discussed in this section can be declared anywhere in REALbasic. For information on structures, see the section "Structures" on page 312 in the chapter on Modules.

Other Data Types

There are many other data types. You will learn about these in the next chapter.

Changing a Value From One Data Type to Another

There may be times when you need to change a value from one data type to another. This is usually because you want to use the value with something that is designed to work with a different data type. For example, you might want to include a number in the title of a window. The title of a window is a string, not a number. Consequently, if you try to assign a number to the title of a window, REALbasic will display an error message when you run your application. The error will tell you that the two data types are not compatible (they are different). Since the window title is a string, you will need to change the number into a string before you can assign it to the window title.

Fortunately, REALbasic has a built-in function called Str (which stands for String) that can change a number into a string. See the Str function in the *Language Reference* for more information. There is also a built-in function called Val (which is short for Value) that changes strings into numbers. See the Val function in the *Language Reference* for more information.

Assigning Values to Properties

The basic syntax for assigning a value is:

```
objectName.propertyName=value
```

For example, if you have a PushButton named PushButton1, and you want to set its Caption property to "OK", you would use the following code:

```
PushButton1.Caption="OK"
```

You can read this as *change PushButton1's caption property to "OK".* This syntax is used when you want a control in a window to change a property of a control in the same window. This syntax is sometimes called dot syntax since the dot is used to indicate that the property to the right of the dot belongs to the object to the left of the dot.

If you want a control to change a property of a control in another open window, you must include the target window's name (not its title) in the syntax. That syntax is:

```
windowName.objectName.propertyName=value
```

For example, suppose you have two open windows whose names are Window1 and Window2 respectively. You want a PushButton on Window1 to set the value of PushButton1's caption on Window2 to "OK". The syntax looks like this:

```
Window2.PushButton1.caption="OK"
```

If you didn't specify the window, REALbasic would assume you meant the control called PushButton1 in the window that contains the object executing the code. If you specify a window that is not open, REALbasic will open the window and make the change. If you have more than one copy of the window open that contains the control you are trying to change, this syntax won't work because you won't be able to tell REALbasic which copy of the window you are referring to. You will learn how to deal with this issue in the next chapter.

If a control is going to change a property belonging to its own window, the window name is not required. The window name is implicit. For example, if you wanted a PushButton to change its window's title property to "Hello World" when the user clicks it, you would use this syntax:

```
Title="Hello World"
```

In some cases, properties are, in effect, grouped hierarchically. That is, one property gives you access to properties and methods of another object. One example of this occurs with the Canvas control. This control comes equipped with a set of drawing tools that allow you to completely customize the appearance of the control. All the drawing tools are accessible via the Graphics property of the Canvas control. For example, the line;

```
Canvas1.Graphics.DrawRect 0,0,100,50
```

draws a 100 x 50 rectangle starting at the 0,0 coordinates of Canvas1. DrawRect is actually a method belonging to the Graphics class—not the Canvas class. The Graphics property of the Canvas class allows you to access this method.

Using Class Constants to Assign a Value

In some cases you need to choose a value from a list of specific values. For example, the Serial class has properties that specify the Baud rate, number of Stop bits, and Parity for serial communications. The Baud rate should be chosen from the list of possible values. The *Language Reference* gives the specific values that each of these properties can accept. In each instance, it also gives class constants for those values that are equivalent to the numeric values. Your code will be much more readable if you use the class constants instead of the numeric values. For example, here are the class constants for specifying the Baud rate.

Baud Rate	Value	Constant	Baud Rate	Value	Constant
300	0	Baud300	9600	8	Baud9600
600	1	Baud600	14400	9	Baud14400

Baud Rate	Value	Constant	Baud Rate	Value	Constant
1200	2	Baud1200	19200	10	Baud19200
1800	3	Baud1800	28800	11	Baud28800
2400	4	Baud2400	38400	12	Baud38400
3600	5	Baud3600	57600	13	Baud57600
4800	6	Baud4800	115200	14	Baud115200
7200	7	Baud7200	230400	15	Baud230400

To refer to a class constant, use the syntax:

```
ClassName.ClassConstant
```

For example, the expression Serial.Baud19200 returns the value of 10.

If you want to assign a baud rate of 57600 to the Serial control named Serial1, you can write:

```
Serial1.Baud=Serial.Baud57600
```

instead of

```
Serial1.Baud=13
```

When you use the class constant, the line of code is much more readable. No one has to go and look up what the 13 means.

Me and *Self*

The REALbasic language also has two very useful pronouns, *Me* and *Self*. *Me* refers to the control that you are working with. For example, if you are trying to set the Caption property of PushButton1 to "OK" in one of the control's event handlers, you could write either:

```
PushButton1.Caption="OK"
```

or

```
Me.Caption="OK"
```

The advantage of using Me is that it makes your code generic. You could use it with any other PushButton control and it would work without change.

The pronoun *Self* refers to the control's parent window. You can use it in an event handler for any control within the window. For example, the lines:

```
FindDialog.Title="Find"
```

and

```
Self.Title="Find"
```

and

```
Title="Find"
```

are all equivalent. In the first example, the Name property of the window is used; that is, the name of the window is FindDialog. In the second example, the Self pronoun accomplishes the same thing without 'hardcoding' the window's name. The last example takes advantage of the fact that the parent window is assumed when you are coding within one of the window's controls.

Getting Values From Properties

You can get a value from a property in almost the same way you store values in properties. The only difference is that the target of the value (where you want the value of the property stored) goes on the left side of the equals sign and the object and property names go on the right. For example, if you had a variable named X and you wanted to assign PushButton1's caption to it, the syntax would be:

```
x=PushButton1.Caption
```

And just as in setting properties, you can get the property of a control in another window by including the window's name. For example, if you want to assign the variable x to the Caption property of PushButton1 in Window2, you would use this syntax:

```
x=Window2.PushButton1.Caption
```

And just like setting properties, if you include only the property name, REALbasic assumes you are referring to a property of the window that contains the control that is executing the code. For example, if you have a PushButton called PushButton1 and you want it to assign the window title to the variable x when it is clicked, you would use this syntax:

```
x=Title
```

Getting and Setting Values in Variables

When you need to store a value that is not associated with an object (the way a property is associated with a control or window), you use a variable. A variable is nothing more than a location in memory that stores a value. Variables have names just like properties do. However, you always have to define a variable and give it its name. The name you give a variable should describe the purpose of the variable. Suppose you want to calculate the age of a person in days from the year he was born. You might have a variable called "Days" to keep track of that information. Variable names can be any length but must begin with a letter or an underscore and can con-

tain only alphanumeric characters (A-Z, a-z, 0-9) or an underscore. Variable names are case-insensitive so REALbasic sees x and X as the same variable.

You can store values in variables and get values from variables in the same way you do with properties. To get a value from a variable, it must be on the right side of the assignment operator (=). Suppose you want to set the caption of a PushButton to the value in a variable called "ButtonTitle".

The first thing you do is create the variable and give it a data type. Like the built-in properties of objects, variables have data types. Before you can use a variable, its data type must be made known using the Dim statement. Dim is short for Dimension, which means to make space for the variable in the computer's memory. You use the Dim statement to declare the name and data type of the new variable. For that reason, it is often called a declaration statement.

In this example, you'd use the statement:

```
Dim ButtonTitle as String
```

This creates the variable "ButtonTitle" in memory and tells REALbasic that it can only store a string value. It can store one string, not several distinct strings.

After you've created it, you can give it a value and assign its value to another variable or a property. For example, you can write:

```
ButtonTitle="Save"
```

After it has a value, you can then assign its value to a property of an existing object. That is, you can assign it to a property that is also of type String.

The example below accomplishes that:

```
PushButton1.Caption=ButtonTitle
```

Conversely, if you wanted to store the value of a property (like the PushButton's caption in the last example) in a variable, you would simply reverse the syntax:

```
Buttontitle=PushButton1.Caption
```

In the example below, the variable i is dimensioned (or "dimmed") as an integer:

```
Dim i as Integer
```

If you have several variables of the same type, you can declare them all with one Dim statement:

```
Dim i,j,k as Integer
```

If you want to declare variables of different types, you can also declare them in one Dim statement. For example, the following Dim statement is valid:

```
Dim Name,Address as String, ShoeSize as Single
```

When you create a variable with the Dim statement you can also assign it a value. You do so by following its data type with an equals sign and the value. For example, the following are valid statements:

```
Dim i as Integer =1
Dim Name as String = "Igor",Address as String = "15 Rue de Vallee"
```

You can also mix variables with and without initial values, as in:

```
Dim a as Integer, b as Integer = 15
```

In this case, the variable a is declared as an integer but gets no value. If you examine the value of a, it will be zero. The variable b, on the other hand, gets the value of 15.

Notice also that the following is valid:

```
Dim a,b,c as Integer = 15
```

This statement declares three integer variables and assigns all of them the value of 15. Although this is valid, you should be careful about assigning values to more than one variable in a single Dim statement because you could easily lose track of some of the variables.

Just like properties, you can only assign values to variables that are compatible with the variable's data type. The last line of the following example generates an error because the types don't match:

```
Dim x,z as Integer
Dim y as String
x=1
y="1.75" //this is the string “1.75” not the number 1.75
z=x+y //error here
```

In the example above x is a number but y is typed as a String. An error is generated because you can't add different data types together. When you try to compile a project that contains this code, REALbasic will stop the compilation process and report that it has found a Type Mismatch Error. It will show you the line of code that caused the error. You cannot test the project until you fix such errors.

The Scope of Variables

A variable that has been declared via the Dim statement exists in memory as long as the particular method is running. It can be accessed only within the method in which it was declared. When the method is finished, the variable is no longer accessible and the memory that was used to store the value becomes available for

other uses. This means that another method in the application cannot access the value of the variable.

For that reason, variables that are declared with the Dim statement in any method are considered *local variables*. They exist only locally, inside the "neighborhood" of the method in which they are declared. That is, the *scope* or access scope of the variable is local.

A Dim statement can be placed anywhere in a method, as long as it precedes the first usage of the variables that it dimensions. If you place a Dim statement inside an If statement, its scope is limited to that If statement, not the entire method. For example, you can write:

```
If x > 5 then
  Dim y as Integer
  y = 10
end if
//y goes out of scope here; the variable name y can now be reused.
//It is redeclared as a string in the following If statement
If x < 5 then
  Dim y as String
  y = "hello"
End if
```

If you need a variable whose scope is greater than local, you need to create a property of an object or a module. Its scope should match the size of the "neighborhood" that needs to read and/or write the value of the property. For example, if the variable needs to be available to some of the controls in a window, consider making it a property of the window. A property that you create can have one of four possible levels of scope:

- **Global:** The property is accessible throughout the application. Global properties can be created only in modules. For more information, see the section "Scope of a Module's Methods, Properties, and Constants" on page 299.
- **Public:** The property is accessible throughout the application. Within the object that owns the property, it is accessible by name only; outside the object, with the syntax *objectName.propertyName*. For example Public property of a window named Window1 would be referred to outside the window as Window1.*PropertyName*, where *PropertyName* is its name. Inside Window1, it can be referred to by name only: *PropertyName*.
- **Protected:** The property is accessible only within the object that owns it, by name only. If you try to access the property outside the object that owns it, REALbasic will display an informative error message.
- **Private:** The property is accessible only within the object that owns it. Other objects based on the owning object cannot access the property (This is not true of

Protected properties). If you try to access the property outside the object that owns it, REALbasic will display an informative error message.

Generally, you will want to choose a scope that is as narrow as possible. This eliminates the possibility that some code outside the object that owns the property might inadvertently read or set the value of the property. For more information about scope, see the section "The Scope of a Property" on page 253.

Declaring Objects

You already know about the data types Integer, String, Boolean, Single, Double, and Color. But variables can also be declared as specific object types. For example, REALbasic has an object called Date, which you use to store dates and times.

You might think that REALbasic should be storing a date in an ordinary variable, but it actually makes sense to make it an object that has several properties. Technically, "Date" is a REALbasic class. You will learn more about classes in Chapter 9.

When you want to use a Date variable, you start with the built-in class as a template for your variable. A template comes with places to store properties of the object and, sometimes, its own methods that perform operations related to the class of objects.

You then create an instance of the class for your use. The instance is the variable that you refer to in your code, not the class itself.

For example, the Date class has separate properties for the Year, Month, and Day, as well as Minute and Second. It has additional properties that format the date as a ShortDate, AbbreviatedDate, or a LongDate.

When you create a new variable based on a class, you start with the same syntax as you use for creating other types of variables. For example, the statement:

```
Dim BirthDate as Date
```

creates a new Date object. However, this statement does not create the instance of the Date template for your use. For example, if you tried to set a property of the variable, BirthDate, to a value, REALbasic wouldn't allow you to do it. For example, Year is an Integer property of a Date object, but you can't read or set it yet. For example:

```
Dim BirthDate as Date
BirthDate.Year=1996
```

doesn't work.

Before you can access or set any of the Date variable's properties, you need to use the New operator to create an instance of the Date object. In programming lingo, creating an instance of an object is called *instantiating* the object.

The following two lines creates and instantiates a Date object:

```
Dim BirthDate as Date
BirthDate=New Date
```

Now, you are ready to access or set any of the Date variable's properties. For example, you can use assignment statements to establish the value of the date.

```
Dim BirthDate as Date
BirthDate=New Date
BirthDate.Year=1996
BirthDate.Month=6
BirthDate.Day=23
```

If you were to access the ShortDate property of the BirthDate variable, it would now be: "6/23/96".

There is a shortcut syntax for declaring and instantiating objects. You can place the New operator in the Dim statement itself. Using this shortcut, you can rewrite this example as:

```
Dim BirthDate as New Date
BirthDate.Year=1996
BirthDate.Month=6
BirthDate.Day=23
```

If you omit the instantiation step, the REALbasic compiler will issue an error message when you first try to read or set one of the object's properties or run one of its methods.

Using Arrays

An array is a special type of variable that contains several values of the same data type. Each variable in the array is called an *element* of the array. To refer to a particular element, you use an *index* number.

The Dim statement also lets you create and type arrays. When you declare an array, you specify the number of elements it has. Later on, you can change the number of elements.

Creating Arrays

You declare an array by specifying the index of the last element of the array. The number of values that you specify in the Dim statement is actually one less than the number of elements in the array because REALbasic arrays always have an element zero. Such an array is sometimes referred to as a *zero-based array*. If the first element of an array is element 1, then it would be called a one-based array.

Since REALbasic arrays are zero-based, the statement:

```
Dim aNames (10) as String
```

creates a string array with eleven elements.

The statement

```
Dim aNames (0) as String
```

creates a string array with one element, element zero. You can read and write to this element just as with any element with a positive index.

In other words, you declare a variable as an array simply by adding the index of the last element to the Dim statement.

If you don't know the size of the array you need at the time you declare it, you can declare it as a null array, i.e., an array with no elements. You do this by using an index of -1 in the Dim statement or leave empty parentheses. This means "an array of no elements." For example, the statements:

```
Dim aNames (-1) as String
```

and

```
Dim aNames () as String
```

creates the array aNames with no elements. Later on, you can resize the array using the ReDim statement or "grow" it element-by-element using the Append or Array methods.

You can create multi-dimensional arrays in REALbasic. For example, a spreadsheet layout can be thought of as a two-dimensional array, rows by columns.

Each dimension is referred to by its own index. For example, the elements of an array with two dimensions are referred to by one index for the rows and the other for the columns. The first element in the upper-left corner is element 0,0.

You create a multi-dimensional array by specifying an index for each dimension. For example, the statement:

```
Dim aNames (2,10) as String
```

creates a two-dimensional array with 3 rows and 11 columns.

You can use constants in Dim statements to set the size of an array. For example, the following declaration refers to a global constant, nLanguages, that is defined in a module:

```
Dim aControl(10,nLanguages) as String
```

In this example, "nLanguages" is the number of languages supported by the application and is used in numerous places in the application. When it changes, you only need to change its definition in the module.

For information on creating constants, see the section "Adding Constants to Modules" on page 305.

Referring to Array Elements

You refer to an element of an array by placing the desired element in parentheses. For example, the statement:

```
aNames(1,1)="Frank"
```

places the string "Frank" in array element (1,1)

The Ubound function returns the index of the last element of a one-dimensional array. For example, the expression UBound(aNames) returns this value. The number of elements is one greater than this number, since the array has an element zero.

For multi-dimensional arrays, Ubound returns the index of the last element of the dimension you specify, or, if you do not specify a dimension, it returns this value for the first dimension. The first dimension is numbered 1. For example, the following example returns 5 in the variable i and 3 in the variable j.

```
Dim i,j as Integer
Dim aNames (5,3) as String
i=Ubound(aNames)
j=Ubound(aNames,2)
```

Initializing Arrays

After you have declared an array, you can assign initial values to the elements with the Array function as well as individual assignment statements, such as shown above. The Array function takes a list of values and assigns the values to the elements of the array, beginning with element zero. In other words, it provides the same functionality as separate assignment statements for each element of the array.

For example, the following statements initialize the array using separate assignment statements for each array element:

```
Dim aNames(2) as String
//using separate assignment statements
aNames(0)="Fred"
aNames(1)="Ginger"
aNames(2)="Stanley"
```

The following statements use the Array function to accomplish the same thing. Note that you don't have to declare the exact size of the array in the Dim statement. The Array function will add elements to the array as needed.

```
Dim aNames(-1) as String
aNames=Array("Fred","Ginger","Stanley")
```

If you declare the array as a fixed size but don't specify as many values as elements, The Array function will start with element zero and use as many elements as are specified.

Array Assignment

If you have two arrays of compatible data types, you can assign one array to the other array. Simply use the assignment statement without the parentheses. Here is a simple example:

```
Dim aNames(2),bNames(2) as String
aNames=Array("Fred","Ginger","Stanley")
bNames=aNames
```

The last statement assigns the values of all three elements of aNames to the first three elements of bNames. If bNames had fewer elements than aNames, then additional elements would first be added to bNames and the assignment of all the elements of aNames to bNames would be completed. For example, the following is valid:

```
Dim aNames(3),bNames(2) as String
aNames(0)="Fred"
aNames(1)="Ginger"
aNames(2)="Tommy"
aNames(3)="Woody"
bNames=aNames
```

After the code runs, the bNames array has a fourth element for storing the value "Woody".

Resizing Arrays

Several methods in the language resize arrays.

- **Append:** The Append method adds an element to the array, increasing its size by one. You pass the value you want to add to the array when you call Append. For example, the following statement:

```
aNames.Append "Dave"
```

adds an element to the array aNames and sets the value of this element to the string, "Dave".

Append works on one-dimensional arrays only.

- **Insert:** The Insert method creates an additional element and inserts it in the place you specify. It takes two parameters, the index of the element to be inserted and the value of the new element. For example:

```
aNames.Insert 9,"Hal"
```

After this statement runs, the value of aNames(9) would be "Hal"; the old element(9) would be shifted up to element (10) and so forth. The size of the array would be increased by one, as with Append.

Insert also works only on one-dimensional arrays.

- **Remove:** The Remove method deletes the element whose index you specify. For example:

```
aNames.Remove 9
```

removes the element with index 9, decreasing the size of the array by one and shifting the array elements after the removed element down by one. It also works on one-dimensional arrays only.

- **Redim:** The Redim statement resizes an existing array. You pass the new values of the array's indexes but you don't specify the data type, which is set by the initial Dim statement. For example, the statement:

```
Redim aNames (100)
```

resizes the array aNames to 101 elements.

Redim works on both one- and multi-dimensional arrays. For multi-dimensional arrays, you can only resize the existing dimensions; you cannot add or reduce the number of dimensions themselves.

A difference between Dim and Redim is that Dim accepts only integers or constants, while Redim accepts any expression that returns an integer. This includes, for example, a user-written function that returns an integer value. Using this feature, you can dimension your arrays on-the-fly.

If you don't know the size of the array you need at the time you declare it, you can declare it as a null array, i.e., an array with no elements, and use the Redim command to resize it later. If your program needs to load a list of names that the user enters, you can wait to size the array until you know how many names the user has entered. You can write a function to figure out what that number is and use it with Redim or use the Append method to add the required elements to the array one-by-one.

Converting to and from an Array to Variables

Two functions enable you to take a take an array and break it up into separate variables and take a single string variable and convert it into an array.

- **Split**: The Split function takes a String variable and creates a one-dimensional array by dividing the string up into elements. It uses a delimiter—a character or character string that signals the end of one element and the start of the next element—to do this task. By default, the delimiter is a space, but you can specify another delimiter.

Here is an example that divides up the contents of a string into array elements. It specifies the comma as the delimiter.

```
Dim aNames(-1) as String //resize using the Split method
Dim s as String
s="Juliet,Taylor,Casting"
aNames=Split(s,",")
```

After this call, the resulting array, aNames, will have three elements:

```
aNames(0)="Juliet"
aNames(1)="Taylor"
aNames(2)="Casting"
```

- **Join**: The Join function takes a one-dimensional array and creates a String variable that contains all the elements in the array, separated by a delimiter character or character string. By default, the delimiter character is a space, but you can specify your own delimiter by passing the delimiter as the second parameter.

  For example, if you have an array that contains elements that store a person's first and last names and phone number, the Join function will create one String variable that contains all the information. Here is an example:

```
Dim aNames(2) as String
Dim s as String
aNames(0)="Anthony"
aNames(1)="Aardvark"
aNames(2)="(406) 737-8946"
s=Join(aNames,",") //creates "Anthony,Aardvark,(406) 737-8946"
```

  In this example, the call to the Join function specifies the name of the array and the comma as the delimiter. If the call were:

```
s=Join(aNames)
```

  the result would be "Anthony Aardvark (406) 737-8946".

Dictionaries

A *dictionary* is a REALbasic object that is made up of a list of *key-value* pairs. That is, each value is paired with an identifying key. The interesting feature of Dictionaries is that both the key and the value are variants. This means that a dictionary can store a mixture of data types—and that the key doesn't have to simply be an integer. You can look up a value in a dictionary by specifying either its key or its sequential position in the dictionary. For example, the *Language Reference* entry for Dictionary has an interesting example in which colors are used as keys.

Mathematical Operators

Performing mathematical calculations is a very common task in programming. REALbasic supports all of the common mathematical operations.

Operation Performed	Operator	Example
Addition	+	2 + 3 = 5
Subtraction	-	3 - 2 = 1
Multiplication	*	3 * 2 = 6
Floating Point Division	/	6 / 4 = 1.5
Integer Division	\	6 \ 4 = 1
Modulo	Mod	6 Mod 3 = 0 6 Mod 4 = 2
Exponentiation	^	2^3 = 8

There are also many built-in mathematical functions. See the *Language Reference* for more information.

REALbasic supports standard mathematical precedence. This means that equations surrounded by parentheses are handled first. REALbasic will begin with the set of parentheses that is embedded inside the most other sets of parentheses. Next, exponentiation is performed, followed by any multiplication or division from left to right. Finally any addition or subtraction is performed. In the example below, the three expressions return different results because of the placement of parentheses:

Expression	Result
2+3*(5+3)	26
(2+3)*(5+3)	40
2+(3*5)+3	20

Constants

A constant is like a variable but it holds a fixed value for its entire "life." When you create a constant, you give it its value. You can read the constant's value in your code, but you cannot use an assignment statement to change the value of a constant. You cannot create array constants.

You can create constants in REALbasic for windows, modules, and objects based on classes that are added to the Project Editor (You'll learn more about windows, modules, and classes later on in this manual.) You can also create a local constant inside any method you write.

Each constant has a Scope. The Scope determines which parts of your application can "see" the constant and read its value. When you create a constant in any method, its scope is automatically Local to that method: Only the code within that method can read the value of the constant. A Local constant is assigned its value within a method and can be referred to anywhere within that method. If another method has a constant with the same name, it is, in effect, a completely different constant.

To define a local constant, use the keyword **Const** within a method, followed by an assignment statement. That is,

```
Const <constname> = <value>
```

You do not have to indicate the data type of a constant. For example, the following statement is valid:

```
Const BackGroundColor=&cFF0000
```

REALbasic will "figure out" that BackGroundColor is a Color. You can then assign the constant BackGroundColor to any property or variable that accepts a Color data type.

The following code is acceptable:

```
Const Accept="OK"
BevelButton1.caption=Accept
```

When the application runs, the PushButton's caption reads "OK".

The Scope of Constants

The Const statement has local scope, just like the Dim statement (see the section "The Scope of Variables" on page 180). When you declare a constant inside a method (as has just been described), it exists only as long as the method is running and can be accessed only inside that method. Other methods can't read its value. It means that the constant is local to the method in which it is declared.

You can also declare constants for windows, modules, or classes. You will learn about managing these objects later in the *Users Guide*.

Reserved Words

The following words should not be used as variable names because they are used as part of the REALbasic language itself:

And	Loop
Array	Me
As	Mod
Boolean	Module
ByRef	Namespace
ByVal	New
Call	Next
Case	Nil
Catch	Not
Class	Object
Color	Of
Const	Or
DebugPrint	Private
Declare	Protected
Dim	Public
Do	Raise
Double	Redim
Downto	Rem
Each	Return
Else	Return
ElseIf	Select
End	Self
Event	Shared
Exception	Single
Exit	Static
False	Step
Finally	String
For	Sub
Function	Then
GoTo	To
Handles	True
If	Try
Implements	Until
In	Wend

Inherits	While
Inline68K	
Integer	
Interface	
Isa	
Lib	

Executing Instructions with Methods

A *method* is one or more instructions that are performed to accomplish a specific task. REALbasic has many built-in methods. For example, the Quit method causes your application to quit. Most REALbasic classes have built-in methods. For example, the ListBox class has a method called AddRow for adding rows to a ListBox (as the name implies).

You can also create your own custom methods. Just like variables, methods are given names to describe them and the same rules apply: the name can be any length, but must start with a letter or an underscore and can contain only alphanumeric values (a-z, A-Z, 0-9) or the underscore character.

The following is an example of a simple method that calculates how many days old a person is in 1998 who was born in 1960:

```
Dim yearBorn, thisYear, daysOld as Integer
yearBorn=1960
thisYear=1998
daysOld=(thisYear-yearBorn)*365
```

Methods can, of course, be far more complex and longer than this example. There are three different places you can put your methods. You will learn about these in the next chapter.

Documenting Your Code

Documenting your code is important because while it might make sense at the time you write it, it may not make sense days or weeks later. You should also name controls and other objects in a logical and consistent way. Also, if someone else has to understand your methods, documentation will make their job a whole lot easier. Comments can be added to your code as separate lines or to the right of any code on an existing line. Comments are ignored by REALbasic when it runs your application and have no impact on performance. In order for the REALbasic compiler to recognize text as a comment, you must start the comment with a single-quote (‘),

two forward slashes (//) or the word REM (short for remark). The example below shows how the previous example could be commented:

```
//Create the necessary variables
Dim yearBorn, thisYear, daysOld as Integer
yearBorn=1960  //set the year they were born
thisYear=1998  //store the current year
//Now calculate the number of days old
daysOld=(thisYear-yearBorn)*365
```

By default, comments in your code appear in red. You can set a different color for comments using REALbasic options (Preferences on Macintosh). For more information, see the section "Configuring the Code Editor" on page 221.

If you have several consecutive lines that you want to convert to comments, click the Comment button in the Code Editor toolbar or use the Edit ▶ Comment menu item (Ctrl+' or ⌘-' on Macintosh). If the line of code that contains the insertion point is a comment, then the Comment button changes to Uncomment and the corresponding menu item changes to Edit ▶ Uncomment. Uncomment changes the lines back to executable code.

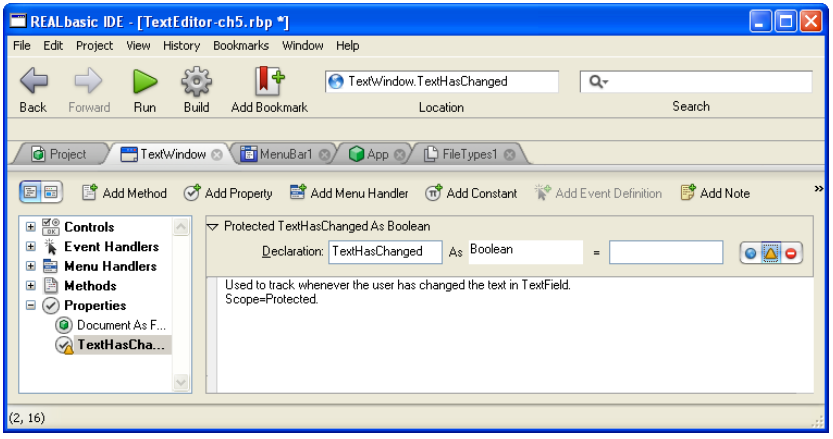
Documenting Properties

When you create a property of a window, class, or module, you can use the Code Editor to document the property. REALbasic automatically states the property's declaration in bold in the Code Editor and you can add any text underneath that heading.

For information about properties, see the section "Adding Properties to Windows" on page 252.

Any text you enter in the code editing area is automatically non-executable, even if it is REALbasic code.

Figure 171. Documenting a window property.

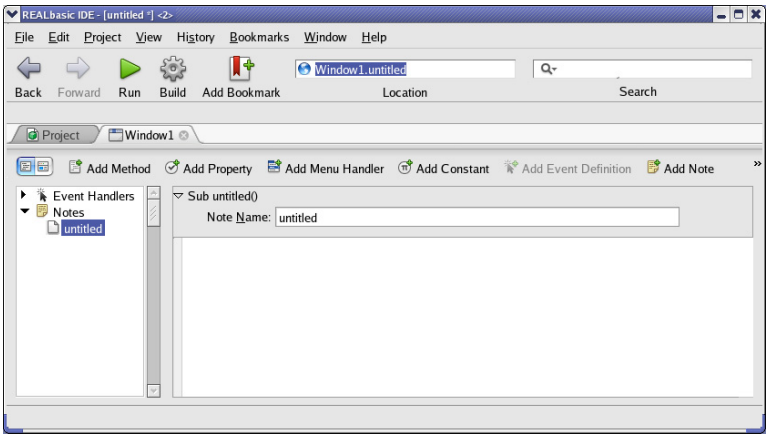


Using Notes

You can also document your code (or application) at a more global level using *Notes*. Each Code Editor has an item called Notes in its browser area that you can use to store comments about your application (or anything else, for that matter). Like comments in your code, Notes are not compiled and are not included in a built application. Although they appear in the Code Editor, you shouldn't try to reference them in your code.

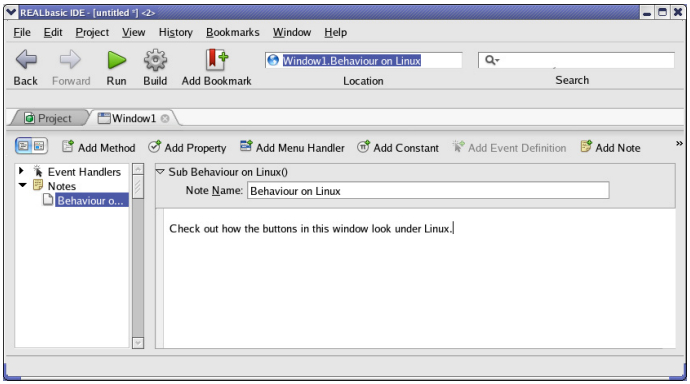
You add a note to a Code Editor just as you would add an object that is compilable. To add a note to a Code Editor, click the Add Note button or choose Project ▶ Add ▶ Note. The Code Editing area changes to an entry area for the note. Enter your note in the entry area.

Figure 172. The Add Note screen.



After you click OK, the name of the note appears in the Notes group and the editing area of the Code Editor opens to that note, enabling you to enter your comments.

Figure 173. The text of a Note.



Passing Values to Methods

Some methods require one or more pieces of information to perform their function. These pieces of information are called *parameters*. Parameters are passed to a method placing them to the right of the method name. In the following example, the AddRow method of a ListBox called ListBox1 is being called. AddRow adds one row to the end of the ListBox and writes text in the new row. In order to do this, you need to pass the text to AddRow as a parameter. Here is an example:

```
ListBox1.AddRow "January"
```

You can also put the parameter in parentheses. The following is equivalent:

```
ListBox1.AddRow ("January")
```

When you pass parameters, you must pass a value of the correct data type. AddRow, for example, requires a String. If you needed to add a number to the ListBox, you must pass it as a String. For example, if you need to add the number 12.7 to the ListBox, you can write:

```
ListBox1.AddRow "12.7"
```

Or, you can use the built-in Str function which converts a number passed to it into a string.The following also works:

```
ListBox1.AddRow Str(12.7)
```

The Str function takes one parameter, a number, and returns a string. So, this expression takes the number you want to display, converts it to a string, and then passes the string to the AddRow method.

If a method requires more than one parameter, use commas to separate them. The ListBox class has a method called InsertRow which is used to insert new rows into a ListBox at any position. The InsertRow method requires two values: the row number where the new row should appear and the text value that should be displayed in the new row. Because more than one parameter is required, the parameters are separated by commas:

```
ListBox1.InsertRow 3, "January"
```

As is the case with one parameter, you can place the list of parameters in parentheses:

```
ListBox1.InsertRow (3, "January")
```

Parameters can also be variables or constants. If a variable is passed as a parameter, the current value of the variable is passed. In the example below, a variable is assigned a value and then passed as a parameter:

```
Dim Month as String
Month="January"
ListBox1.InsertRow 3, Month
```

In the next chapter, you will learn how to define parameters for your own custom methods.

Passing Arrays as Parameters

A one-dimensional array can be passed as a parameter in a call to a subroutine or function. To specify that a parameter is an array, put empty parentheses after its name in the declaration. For example,

```
Names () as String
```

can be used in the declaration when you want to pass an array of strings to the subroutine. Since you do not need to specify the number of elements in the array to be passed, you can pass a different number of elements at different places in your code.

When you pass an array to the subroutine, omit the parentheses. For example, use

```
PrintLabels (Names)
```

where PrintLabels is the name of the method that accepts the string array as its parameter.

Returning Values from Methods

Some methods return values. This means that a value is passed back from the method to the line of code that called the method. For example, REALbasic's built-in method, Ticks, returns the number of ticks (60th's of a second) that have passed since you turned on your computer. You can assign the value returned by a method the same way you assign a value. In the example below, the value returned by Ticks is assigned to the variable x:

```
x=Ticks
```

Some methods require parameters and return a value. For example, the Chr function returns the character whose ASCII code is passed to it. When you pass parameters to a method that returns a value, the parameters must be enclosed in parentheses. In the example below, the Chr function is passed 13 (the ASCII code for a carriage return) and returns a carriage return to the variable x:

```
x=Chr(13)
```

The parentheses are required because the value returned might be passed as a parameter to yet another method. Without the parentheses, it would be difficult to

distinguish which parameters were being passed to which method. In the example below, the numeric value returned by the Len function (which returns the number of characters in the string passed to it) is then passed to the Str function (which converts a numeric value to a string). The string returned by the Str function is then passed as a parameter to the InsertRow method of a ListBox:

```
ListBox1.InsertRow 3, Str(Len("Hello"))
```

Methods that return a value are referred to as *functions*. In the REALbasic *Language Reference*, the names of methods that return a value are followed by the word *function*. In the next chapter, you will learn how to return values from your own custom functions.

Passing Parameters by Value and by Reference

By default, you pass values to a method by value. When you do so, the method receives a copy of the data in the object that you pass. Your method receives the data and can perform operations on it.

Parameters passed by value are treated as local variables inside the method—just like variables that are created using the Dim statement. This means that you can modify the values of the parameters themselves rather than first assigning the parameter to a local variable. For example, if you pass a value in the parameter "x", you can increment or decrement the value of x rather then assigning the value of x to a local variable that is created using Dim.

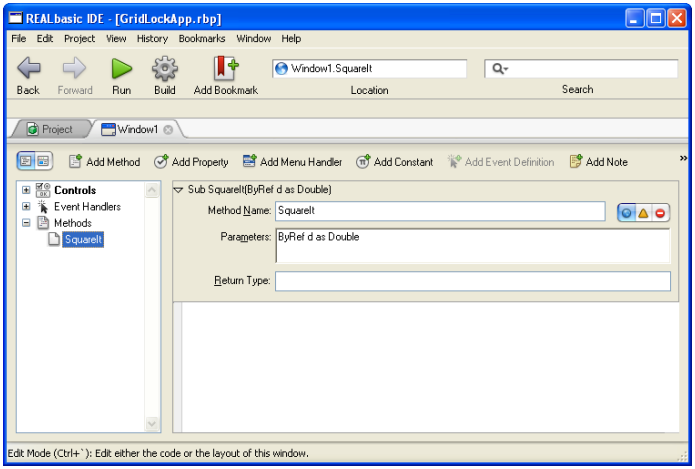
For example, the following method is valid:

```
Sub SquareIt(a As Integer)
  a=a*a
  MsgBox str(a)
```

When you write your own methods, you have the option of passing information by reference. When you pass information by reference, you actually pass a pointer to the object containing the information. The practical advantage of this technique is that the method can *change the values* of each parameter and replace the values of the parameters with the changed values. When you pass parameters by value, you can't do this because the parameter only represents a copy of the data itself.

You use the keywords ByVal or ByRef to specify the type of parameter passing (ByVal is assumed and does not need to be used). To pass a parameter by reference, use the ByRef keyword in the method declaration when you declare a parameter that is to be passed by reference. For example, Figure 174 shows a parameter that is declared ByRef. The method can replace the parameter with the computed value.

Figure 174. Declaring a parameter ByRef.



Suppose the code that calls this method is:

```
Dim a as Integer
a=3
SquareIt a
EditField1.text=str(a)
```

and the method is simply:

```
a=a*a
```

The EditField will display the number 9. If you don't specify ByRef, the method can change the value passed to it, but it cannot return the changed value.

Arrays are passed by reference. The method can change all or some of the elements of the array.

When you want to use parameter passing by value, you do *not* need to use the ByVal keyword explicitly. Parameter passing by value is the default and is used unless overridden by use of ByRef.

Comparison Operators

There are many times when you need to compare two values to determine whether or not a particular condition exists. When making a comparison, what you are really doing is making a statement that will either be True or False. For example, the statement "My dog is a cat" evaluates to False. However, the statement "My dog

weighs more than my cat" may evaluate to True. The table below shows examples of the comparison operators that are available:

Description	Symbol	Numeric Example	Evaluates To
Equality	=	5=5	True
Inequality	<>	5<>5	False
Greater Than	>	6>5	True
Less Than	<	6<5	False
Greater Than or Equal To	>=	6>=5	True
Less Than or Equal To	<=	6<=5	False

String and boolean values can also be used for comparisons. String comparisons are case insensitive and alphabetical. This means that "Jeannie" and "jeannie" are equal. But "Jason" is less than "Jeannie" because "Jason" falls alphabetically before "Jeannie". If you need to make case sensitive or lexicographic comparisons, see the StrComp function in the *Language Reference*.

Testing Multiple Comparisons

You can test more than one comparison at a time using the And, Or, and Not operators.

And Operator

Use this operator when you need to know if all comparisons evaluate to True. In the example below, if the variable x is 5 then the expression evaluates to False:

```
x>1 And x<5
```

Or Operator

Use this operator when you need to know if any of the comparisons evaluate to True. In the example below, if the variable x is 5 then the expression evaluates to True:

```
x>1 Or x<5
```

Not Operator

Use the Not operator to reverse the value of a boolean variable. For example:

```
Not (x < 0)
```

tests whether x is equal to or greater than 0.

Executing Instructions Repeatedly with Loops

There may be times when one or more lines of code need to be executed more than once. If you know how many times the code should execute, you could simply repeat the code that many times. For example, if you wanted a PushButton to beep three

times when clicked, you could simply put the Beep method in your code three times like this:

```
Beep
Beep
Beep
```

But say you need it to beep fifty times or perhaps until a certain condition is met? Simply repeating the code over and over in these cases will either be just tedious or not possible. How do you solve this problem? The answer is a *loop*.

Loops execute one or more lines of code over and over again.

While...Wend

A While loop executes one or more lines of code between the While and the Wend (While End) statements. The code between these statements is executed repeatedly, provided that the condition passed to the While statement continues to evaluate to True. Consider the following example:

```
Dim n As Integer
While n<10
  n=n+1
  Beep
Wend
```

The variable "n" will be zero by default when it is created by the Dim statement. Because zero is less than ten, execution will move inside the While...Wend loop. The variable n is incremented by one. The Beep method plays the alert sound. REALbasic checks to see if the condition is still True and if it is, then the code inside the loop executes again. This continues until the condition is no longer True. If the variable n was not less than ten in the first place, execution would continue at the line of code after the Wend statement.

Do...Loop

Do loops are similar to While loops but a bit more flexible. Do loops continue to execute all lines of code between the Do and Loop statements *until* a particular condition is True. While loops on the other hand execute as long as the condition remains True. Do loops provide more flexibility than While loops because they allow you to test the condition at the beginning or end of the loop. The example

below shows two loops; one testing the condition at the beginning and the other testing it at the end:

```
Do Until n=10
  n=n+1
  Beep
Loop

Do
  n=n+1
  Beep
Loop Until n=10
```

The difference between these two loops is this. In the first case, the loop will not execute if the variable n is already equal to ten. The second loop executes at least once regardless of the value of n because the condition is not tested until the end of the loop.

It is possible to create a Do loop that does not test for any condition. Consider this loop:

```
Do
  n=n+1
  Beep
Loop
```

Because there is no test, this loop will run endlessly. You can call the Exit method to force a loop to exit without testing for a condition. However, this is poor design because you have to read through the code to figure out what will cause the loop to end.

Endless Loops

Make sure that the code inside your While and Do loops eventually causes the condition to be satisfied. Otherwise, you will end up with an endless loop that runs forever. Should you do this accidently, you can switch back to the IDE by clicking the Stop button in the Run screen or closing the Run screen in the IDE to stop the loop. If this doesn't work, you will need to force REALbasic to quit by pressing Ctrl+Alt+Del or ⌘-Option-Escape (Macintosh).

Lengthy Loops

When a loop starts running, its process 'takes over' and doesn't allow the user to interact with interface elements such as menus, buttons, and scroll bars. On modern computers and reasonably short loops, this isn't a problem because the loop executes faster than the user can think of another button to push or menu item to select. If this is not true, there are a couple of things you can do:

- If the user should wait until the loop is finished before doing anything else (e.g., if a user action might invalidate the results of the loop), you can signal that a lengthy operation is in progress by changing the mouse cursor to a watch cursor until the

loop ends. Keep in mind that, with this solution, the user can't even stop the loop prematurely because the loop has 'taken over' the application. See the section on the MouseCursor class in the *Language Reference* for more information.

- If the user is permitted to do other tasks while the loop is running, you should place the code for the loop in a separate thread. A thread runs as a background task, allowing user operations in the foreground. Please see the entry in the *Language Reference* for the Thread class for information on placing code in threads.

  Be aware of lengthy operations that take place inside loops. When programming a lengthy operation, you can use the DoEvents method of the Application class inside the operation to yield time back to REALbasic so that it can handle other events, such as any user input from the keyboard or mouse. See the entry for the Application class in the *Language Reference* for more information.

For...Next

While and Do loops are perfect when the number of times the loop should execute cannot be determined because it's based on a condition. A For loop is for cases in which you can determine the number of times to execute the loop. For example, suppose you want to add the numbers one through ten to a ListBox. Since you know exactly how many times the code should execute, a For loop is the right choice. For loops also differ from While and Do loops because For loops have a loop counter variable, a starting value for that variable and an ending value. The basic construction of a For loop is:

```
Dim Counter As Integer
For counter=0 to 100
 // [your code goes here]
Next
```

Notice that the Dim statement declares the counter as an Integer. This is the most common way to define the counter, but it is not required. You can also declare the counter variable as a Single or Double.

In this example, the counter variable was declared in the usual way, via the Dim statement. Since counter variables are rarely needed outside the For loop, REALbasic also allows you to declare the counter variable right inside the For statement. In other words, you can redo this example in the following way:

```
For Counter As Integer=0 to 100
  // [your code goes here]
Next
```

Notice that the Dim statement has been removed from the example. If you declare the counter variable this way, you can use it only within the For loop. It disappears after the For loop is finished. If you need to read or change the value of the counter variable outside the For loop, you should use the Dim statement instead.

In this case, the starting value and the ending value are specified as numbers. You can also use variables, as shown in this example:

```
Dim StartingValue, EndingValue As Integer
StartingValue=0
EndingValue=100
For counter As Integer=StartingValue to EndingValue
  [your code goes here]
Next
```

The first time through the loop, the counter variable will be set to StartingValue. When the loop reaches the Next statement, the counter variable will be incremented by one. When the Next statement is reached and the counter variable is equal to endingValue, the counter will be incremented and the loop will end.

Let's take a look at the example mentioned earlier. You want to add the numbers one through ten to a ListBox. The following example accomplishes that:

```
For i as Integer=1 to 10
  ListBox1.AddRow Str(i)
Next
```

The counter variable (i in this case) is passed to the Str function to be converted to a string so that it can be passed to the AddRow method of ListBox1.

Note: The letter "i" is commonly used as the loop counter for historical reasons. In FORTRAN, the letters I to N are typed as integers by default. Therefore, FORTRAN programmers began the practice of using those letters as counters, and in the order they appear in the alphabet. That is, if a FORTRAN programmer needed to nest one loop in another (as is described on page 204), he would use j as the counter for the inner loop. This convention made it easy for FORTRAN programmers to follow the logic of code that processed multi-dimensional arrays.

By default, For loops increment the counter by one. You can specify another increment value using the *Step* statement. In this example, the Step statement is added to increment the counter variable by 5 instead of 1:

```
For i as integer=5 to 100 Step 5
  ListBox1.AddRow Str(i)
Next
```

In this example, the For loop starts the counter at 100 and decrements by 5:

```
Dim i As Integer
For i=100 DownTo 1 Step 5
 ListBox1.AddRow Str(i)
Next
```

So far, we have looked at cases where *StartingValue* and *EndingValue* are integer numbers. If either *StartingValue* or *EndingValue* are expressions that must be evaluated to integers, the For loop will perform the evaluation each time it increments the counter — even if the expression always evaluates to the same integer.

Therefore, it is advisable to perform any evaluations before entering the loop. For example, consider a loop that needs to process all the fonts that are installed on the user's computer. This number cannot be known in advance but there is a built-in function in REALbasic, FontCount, that you can use to obtain the total number of fonts. If you use it in the For statement to compute *EndingValue* (like so):

```
For i as Integer=0 to FontCount-1
.
.
Next
```

The loop will run more slowly than if you calculate the value only once:

```
Dim nFonts as integer
nFonts=FontCount-1
For i as Integer=0 to nFonts
.
.
Next
```

However, the difference in speed may be of no practical value unless it is a very lengthy loop. On the computer that is being used to write this manual, the number of installed fonts is 120 and the difference in speed between these two loops is approximately 1/250 of a second—not enough to lose sleep over.

A For loop (as well as any other kind of loop) can have another loop inside it. In the case of a For loop, the only thing you will have to watch out for is making sure that the counter variables are different so that the loops won't confuse each other. The example below uses a For loop embedded inside another For loop to go through all the cells of a multi-column ListBox counting the number of cells in which the word "Hello" appears:

```
Dim row, column, count As Integer
For row=0 to ListBox1.ListCount-1
  For column=0 to ListBox1.ColumnCount-1
    If ListBox1.Cell(row,column)="hello" then count=count+1
    End if
  Next
Next
MsgBox Str(count)
```

Another way to keep this straight is to use the naming convention started by FORTRAN programmers of using the letters of the alphabet beginning with "i" as

the counters. In that way, you'll always know which loop is inside another loop without trying to figure out what the loops are supposed to be doing.

For loops are generally more efficient than Do and While loops because the compiled code generated is more efficient.

The For…Each statement

Another situation in which you want to loop through a group of values is array processing. Rather than looping through a set of statements for each value of a counter, the For…Each statement processes each element of an array that is passed to it. Here is a simple example:

```
Function SumStuff(values() as Double) as Double
 Dim sum, element as Double
 For Each element In values
  sum=sum+element
 Next
Return Sum
```

"Values" is a one-dimensional array of numbers that is passed to the function SumStuff. In the For Each statement, role of the counter variable is taken by the variable "element", which refers to an element in the array. The For Each loop executes the statements between the For Each statement and the Next statement for each element in the array.

Since the array doesn't necessarily have to be numbers, this statement enables you to process a group of objects of any type. They could be pictures, colors, documents, sets of database records, and so forth.

The For Each statement identifies the counter variable (element) and the array to be processed. To call the function, pass an array of doubles in a statement such as:

```
s=SumStuff(MyNumbers)
```

where s is declared as a Double and MyNumbers is an array of Doubles.

As is the case for the For...Next loop, you can declare the data type of the counter inside the For Each statement rather than in a separate Dim statement. For example the previous example could be rewritten like this:

```
Function SumStuff(values() as Double) as Double
 Dim sum as Double
 For Each element as Double In values
  sum=sum+element
 Next
Return Sum
```

Adding Loops to your code

The Code Editor's contextual menu offers an especially convenient way of adding loops to your code. The last three items in the contextual menu wrap the selected

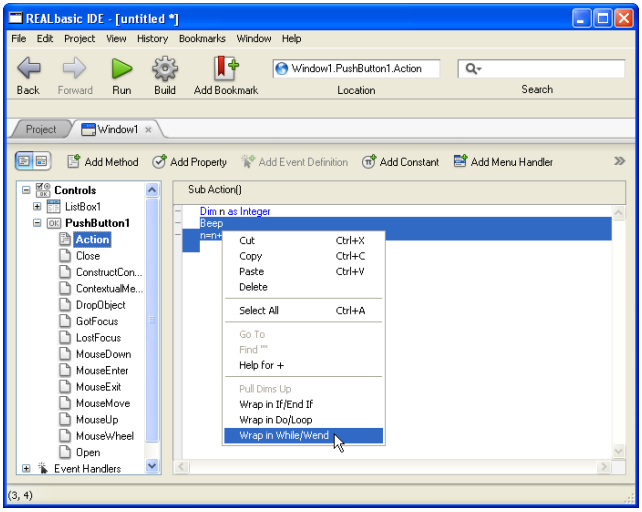
lines of code inside a type of loop. To use these menu items, simply write the code that goes inside the loop, select the lines, and then choose the type of loop from the contextual menu. Your choices are If...End If, Do...Loop, and While...Wend.

To take the example of the While...Wend loop shown on page 200, suppose the user has typed only the lines:

```
Dim n As Integer
n=n+1
Beep
```

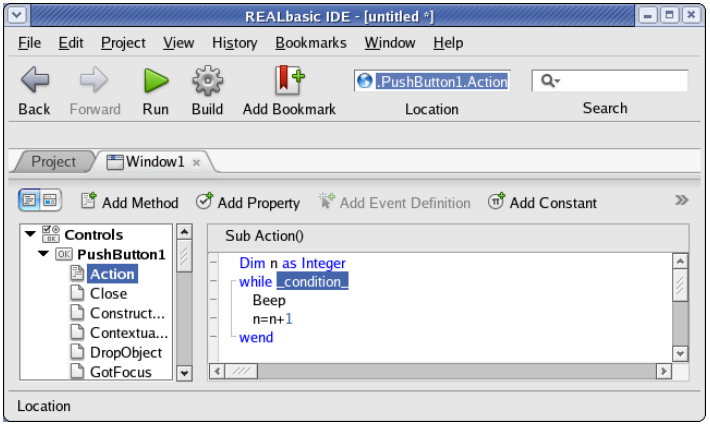
and selected the two lines that go inside the loop before choosing the Wrap in While...Wend contextual menu item.

Figure 175. Wrapping lines of code inside a loop.



The result is shown in Figure 176:

Figure 176. The Code Editor after wrapping lines inside a loop.



REALbasic can wrap the selected lines in the loop but it doesn't know what condition should terminate the loop. Therefore, it has inserted the placeholder text _condition_ in the While statement. It has already been selected, so all you need to do is replace it with the condition you want. In this example, it's n < 10.

This process is the same for the Do...Loop and If...End If loops, described in the next section.

Making Decisions with Branching

The methods you write execute one line at a time from top to bottom, left to right. There will be times when you want your application to execute some of its code based on certain conditions. When your application's logic needs to make decisions it's called *branching*. This allows you to control what code gets executed and when. REALbasic provides two branching statements: If…Then and Select…Case.

If…Then…End If

The If…Then statement is used when your code needs to test a single boolean (True or False) condition and then execute code based on that condition. If the condition you are testing is True, then the lines of code you place between the If...Then line and the End If line are executed.

```
If condition Then
 //[Your code goes here]
End If
```

Say you want to test the integer variable month and if its value is 1, execute some code:

```
If month=1 Then
 //[Your code goes here]
End If
```

month=1 is a boolean expression; it's either True or False. The variable month is either 1 or it's not 1.

Suppose you have a PushButton that performs an additional task if a particular CheckBox is checked. The value property of a CheckBox is boolean so you can test it in an If statement easily:

```
If CheckBox1.value Then
 //[Your code goes here]
End If
```

If...Then...Else ...End If

In some cases, you need to perform one action if the boolean condition is True and another if it is False. In these cases, you can use the optional Else clause of an If statement. The Else clause allows you to divide the code to be executed into two sections: the code that is executed when the condition is True and the code that is executed when it is False. In this example, one message is displayed if the condition is True while another is displayed if it is False:

```
If month=1 Then
  MsgBox "It's January."
Else
  MsgBox "It's not January."
End If
```

If...Then...Else If...End If

In some cases, you need to perform an additional test when the initial condition is False. Use the optional ElseIf statement. In the example below, if the variable month is not 1, then the ElseIf statement performs an additional test:

```
If month=1 Then
  MsgBox "It's January."
ElseIf month<4 Then
  MsgBox "It's still Winter."
End If
```

You could, of course, use an additional If...Then...EndIf statement inside the Else portion of the first If statement to perform another test. However, this adds another EndIf and needlessly complicates your code. You can use as many ElseIf statements as you need.

In this example, another ElseIf has been added to perform an additional test:

```
If month=1 Then
  MsgBox "It's January."
ElseIf month<4 Then
  MsgBox "It's still Winter."
ElseIf month<6 Then
  MsgBox "It must be Spring."
End If
```

If the initial condition is False, REALbasic continues to test the ElseIf conditions until it finds one that is True. It then executes the code associated with that ElseIf statement and continues executing the lines of code that follow the End If statement.

If...Then...Else

An If statement can be written on one line, provided the code that follows the Then and (optionally) the Else statements can all be written on one line. When you use this syntax, you omit the End If statement. For example, the following statements are valid:

```
If error=123 Then MsgBox "An error occured."
If error=123 Then MsgBox "An error occured." Else MsgBox "Success"
If error=103 Then Break //breaks into the debugger
```

#If…#Endif

The #If...#Endif statement is designed to handle a very special case of conditional branching. You use #If...#Endif when you need to compile different versions of code for different platforms. That's its only use. The boolean condition that the #If statement uses will accept only special boolean constants that can determine the type of code that is being compiled. Here is the list of boolean constants that the #If statement accepts.

Boolean Constant	Description
DebugBuild	The application is running within the REALbasic application, i.e., from clicking the Run button in the IDE Main toolbar.
RBVersion	Returns the version of REALbasic that is being compiled. You can use this in an expression that evaluates to True or False to determine which version of REALbasic is compiling the application.
TargetBigEndian	The compiled application is running on a machine that uses the Big Endian byte order. Macintosh use the Big Endian byte order.
TargetCarbon	The compiled application is currently running Carbon/Mac OS X code, i.e., code that runs on both Mac OS X and Mac OS "classic".
TargetHasGUI	The application has a graphical user interface, i.e., it is not a ConsoleApplication or a ServiceApplication.

Boolean Constant	Description
TargetLinux	The compiled application is running Linux code.
TargetLittleEndian	The compiled application is running on a machine that uses the LittleEndian byte order. PCs use the Little Endian byte order.
TargetMachO	The compiled application is running on Mac OS X, running object code in the format for the Mach kernel. This code runs only on Mac OS X.
TargetMacOS	The compiled application is currently running Macintosh code, PPC 'classic', or Mac OS X, in either the MachO or PEF formats.
TargetMacOSClassic	The compiled application is currently running Macintosh code within a "classic" Mac OS supported by REALbasic (Mac OS 8-9).
TargetWin32	The compiled application is currently running Win32 code. It will run on any version of Windows from Windows 98/ME to XP, including Windows NT and Windows 2000.

For example, if you want to use code that manages the Mac OS X dock, you can include it only in your MachO build of the application with a statement such as:

```
#if TargetMachO
   //code goes here
#Endif
```

If you need to include different code for different platforms rather than just including code only for one platform, you can use the #Elseif keyword in this manner:

```
#If TargetWin32
 //Windows specific code here
#ElseIf TargetMacOS
 //Macintosh code goes here.
#ElseIf TargetLinux
 //Linux code goes right here.
#EndIf
```

You can also include or exclude code based on the version of REALbasic that is running. You use the RBVersion function to get the version of REALbasic that's running.

```
#If RBVersion>5
 //include 5.0 code here
#endif
```

This example only includes the code if the version of REALbasic that is compiling the code is at least 5.0.

Note that the "then" keyword is not needed in the #If statement. It is optional.

For more information on conditional compilation, see the entries in the Language Reference for the #If statement and the boolean constants and Chapter 14, "Building Stand-Alone Applications" on page 547.

Select...Case

When you need to test a property or variable for one of many possible values and then take action based on that value, use a Select...Case statement. Consider the following example that tests a variable (dayNumber) and displays a message to the user to tell him which day of the week it is:

```
If dayNumber=2 Then
  MsgBox "It's Monday."
ElseIf dayNumber=3 Then
  MsgBox "It's Tuesday."
ElseIf dayNumber=4 Then
  MsgBox "It's Wednesday."
ElseIf dayNumber=5 Then
  MsgBox "It's Thursday."
ElseIf dayNumber=6 Then
  MsgBox "It's Friday."
Else
  MsgBox "It's the weekend."
End If
```

No two of these conditions can be True at the same time. While this method of writing the code works, it's not that easy to read. In this example, the same code is presented in a Select...Case statement, making it far easier to read:

```
Select Case dayNumber
Case 2
  MsgBox "It's Monday."
Case 3
  MsgBox "It's Tuesday."
Case 4
  MsgBox "It's Wednesday."
Case 5
  MsgBox "It's Thursday."
Case 6
  MsgBox "It's Friday."
Else
  MsgBox "It's the weekend."
End Select
```

The Select...Case statement compares the variable or property passed in the first line to each value on the Case statements. Once a match is found, the code between that case and the next is executed. Select...Case statements can contain an Else statement to handle all other values not explicitly handled by a case.

The Select…Case statement works with variables of any data type. It works for strings, integers, singles, doubles, booleans, and colors. For example, you can compare colors, as in the following example:

```
Dim c as Color
c=&cFF0000 //pure red
Select case c
 Case &c00FF00 //green
  MsgBox "Green"
 Case &cFF0000 //red
  MsgBox "Red"
 Case &c0000FF //blue
  MsgBox "Blue"
End select
```

A Case statement can accept more than one value, with different values separated by commas. For example, the following is valid:

```
Dim c as color
c=&cFF0000 //red
Select case c
 Case &c00FF00,&cFF000 //green, red
  MsgBox "Green or Red"
 Case &cFF0000 //red
  MsgBox "Red"
 Case &c0000FF //blue
  MsgBox "Blue"
End select
```

In the preceding example, the first Case statement is True, so its code executes. Although the color passed to Select…Case is Red, the code for the second case does not execute because it is not the first matching case.

The Select Case statement accepts an “Else” clause. The code in the Else clause executes only if none of the preceding cases matches. The Else clause can be written

as either “Else” or “Case Else”. In the following example, the Case Else clause executes because the color FFFF00 was passed:

```
Dim c as Color
c=&cFFFF00 //pure red
Select case c
 Case &c00FF00 //green
  MsgBox "Green"
 Case &cFF0000 //red
  MsgBox "Red"
 Case &c0000FF //blue
  MsgBox "Blue"
 Case else
 MsgBox "None of the above"
End select
```

The Case statement can also accept a range of consecutive values that you pass using the “To” keyword. For example:

```
Dim i as Integer = 53
Select case i
 Case 1 to 25
  MsgBox "25 or less"
 Case 26 to 50
  MsgBox "26 to 50"
 Case 51 to 100
  MsgBox "51 to 100"
End Select
```

In this example, the third case, “51 to 100”, is true.

You can combine ranges with nonconsecutive values, by separating them with commas, such as:

```
Case 0, 26 to 50, 75, 100 to 200
```

You can write inequalities with the “Is” keyword and an inequality operator. The syntax is:

```
Is ineqalityOperator
```

For example:

```
Dim i as Integer = 10
Select Case i
 Case Is <= 10
//this case selected
 Case Is > 10
//this case not selected
End Select
```

You can combine inequalities with values, as in:

```
Dim i as Integer = 75
Select Case i
Case 0, Is <=10,100
//case not selected
Case Is > 10, Is < 99
//case selected
End Select
```

You can even use functions that return a value of the specified data type in a Case statement. Here is a simple example:

```
Dim i as Integer = 4
Dim a as Integer = 2
Select Case i
 Case Functx(a)
  //case 1
 Case a
  //case 2
Else
  //no match
End Select
```

The function in the first Case statement is:

```
Function Functx(a As Integer) as Integer
 Return a*a
```

In this example, the function squares the value passed to it, so the first Case statement matches.

In the case of a simple function like this, you can write the expression in the Case statement itself. That is, the following is an equivalent matching Case statement:

```
Case a*a
```

The Select Case statement can also compare variables of type Object. The following example uses a Select…Case statement to determine which button the user pressed

in a MessageDialog box. This example was shown in the section "The MessageDialog Class" on page 80 and the resulting dialog box is shown in Figure 57 on page 83. The Select Case statement compares objects of type MessageDialogButton to determine which of three possible dialog buttons was pressed.

```
Dim d as New MessageDialog  //declare the MessageDialog object
Dim b as MessageDialogButton //for handling the result
d.icon=MessageDialog.GraphicCaution   //display warning icon
d.ActionButton.Caption="Save"
d.CancelButton.Visible=True   //show the Cancel button
d.AlternateActionButton.Visible=True   //show the alternate action
button
d.AlternateActionCaption='Don't Save"
d.Message="Save changes before closing?"
d.Explanation="If you don't save your changes, you will lose "_
     +"your work."
b=d.ShowModal   //display the dialog
Select Case b   //the MessageDialogButton returned by d
 Case d.ActionButton    //determine which button was pressed.
   //user pressed Save
 Case d.AlternateActionButton
   //user pressed Don't Save
 Case d.CancelButton
   //user pressed Cancel
End Select
```

CHAPTER 5

Programming with Events and Objects

Most of your code will execute in response to something the user does, such as selecting a menu item, clicking on a button, or typing in an EditField. This kind of programming is called *event-driven programming* because events cause the programming code to execute. Understanding how events work and which user actions cause which events to occur will take you a long way towards getting your application to do what you want it to do.

In this chapter you will learn about event-driven programming, how to use the Code Editor, and how to get your application to respond when the user clicks on interface objects or types on the keyboard.

Contents

Understanding Event-Driven Programming

Your users will interact with your applications by clicking the mouse and typing on the keyboard. Each time the user clicks the mouse on a part of your application's interface or types something in an EditField, an event occurs. The event is simply the action the user took (the mouse click or the key press) and where it took place (on this button, on that menu item, or in this EditField). Some events can indirectly cause other events. For example, when the user selects a menu item (causing an event) that opens a window, it causes another event — the opening of the window).

Each object you create in REALbasic can include, as part of itself, the code you write that executes in response to the various events that can occur for that type of object. For example, a PushButton can include the code you wish to execute when the PushButton is pushed. An object can even respond to events you might not have thought it could — such as responding as the user moves the pointer over a button. When the user causes an event, REALbasic checks to see if the object the event was directed towards has any code that needs to execute in response to that event. If the object has code for the event, REALbasic executes that code and then waits for the user to cause another event to occur. This continues until something causes the application to quit, usually the user's choosing Exit from the File menu (Quit on Macintosh).

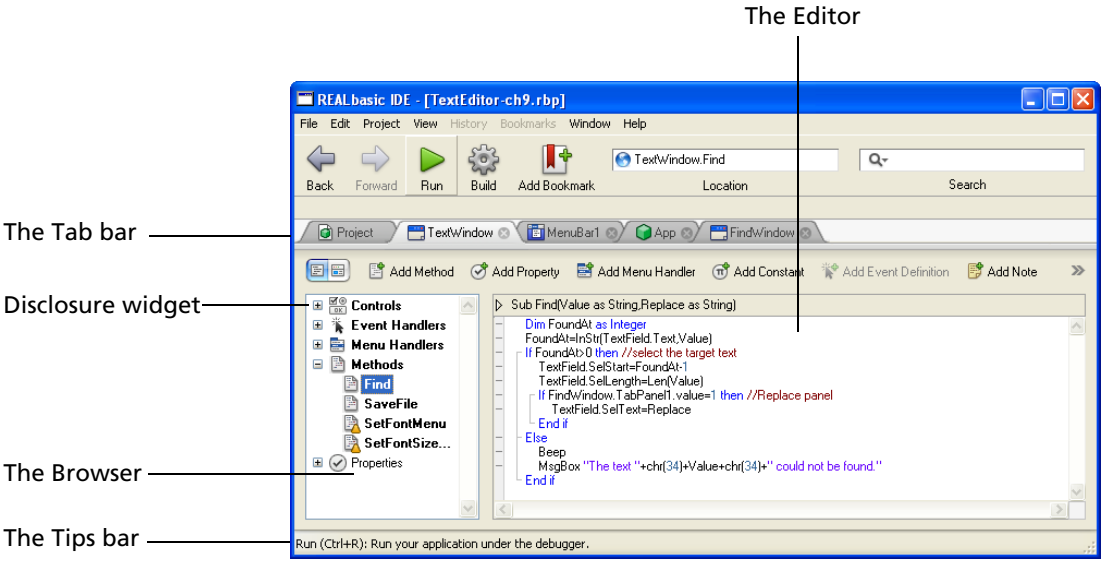
As mentioned earlier, the user can also indirectly cause events to occur. Buttons, for example, have an event called Action which occurs when the user clicks the button. The code that handles the response to an event is called (appropriately enough) an *event handler*. Suppose the button's Action event handler has code that opens another window. When the user clicks the button, the Action event handler opens a window and REALbasic sends an Open event to the window. This is not an event the user caused directly. The user caused this event indirectly by clicking the button whose code opened the new window.

There are many events that can occur to each object in your application. The good news is that you don't have to learn about all of them. You simply need to know where to look for them so that, if you want to respond to an event, you can find out if the object is able to respond to that event. Later in this chapter, you will learn about many of the common events you will need to be aware of in order to create your applications.

Using The Code Editor

You use the Code Editor to enter the code for the various events that can occur for the objects that make up your application's interface. It's also used to add properties and methods to objects. The Code Editor has two sections: the Browser and the Editor itself.

Figure 177. The Code Editor.



The Browser is a hierarchical list of the programming-related components that make up a particular object. For a window, the Browser lists the window's:

- Controls
- Event Handlers
- Menu Handlers
- Methods
- Properties
- Notes

If there are no items in a category, the category does not appear.

You will learn more about each of these items later in this chapter.

Opening the Code Editor

Use the Code Editor to edit the code for controls, windows, classes, and modules. There are several ways to open the Code Editor for a specific window.

To open the Code Editor for a window, do this:

1 Click the tab in the Tab bar belonging to the window's Window Editor or, if it does not yet have a tab, double-click the name of the window in the Project Editor.

By default, the Window Editor for the window appears. The Edit Mode buttons on the left side of the Window Editor toolbar toggle between displaying the Window Editor and the Code Editor for the window

On Windows, the selected icon is in its depressed state; on Macintosh and Linux, the selected icon has the system highlight color.

Macintosh and Linux

Windows

Code Editor Selected

Window Editor Selected

The keyboard equivalents for switching between these two editors is Ctrl+` (Windows and Linux) and Command+` (Macintosh). You can also switch editors with the View ▶ Show Layout and View ▶ Show Code menu items. These menu items are available only when a Window Editor is displayed.

2 Click the Code Editor icon, the keyboard equivalent, or menu command to switch editor views.

The Window Editor switches to the Code Editor for the Window.

If the Project Editor is open, you can go the Code Editor for a window directly by holding down the Ctrl+Shift keys (Windows and Linux) or Option (on Macintosh) when you double-click or press Enter (Return on Macintosh) on the Window's name.

To open the default event in the Code Editor for a specific control, do this:

1 If it is not already open, click on the tab for the window that contains the control.

The Window Editor for the window appears.

2 Double-click on the control.

This will open the Code Editor for the control's parent window. REALbasic will then automatically expand the Controls group in the browser area, expand the control you double-clicked on, and select the default event handler (e.g., the Action event handler for a PushButton).

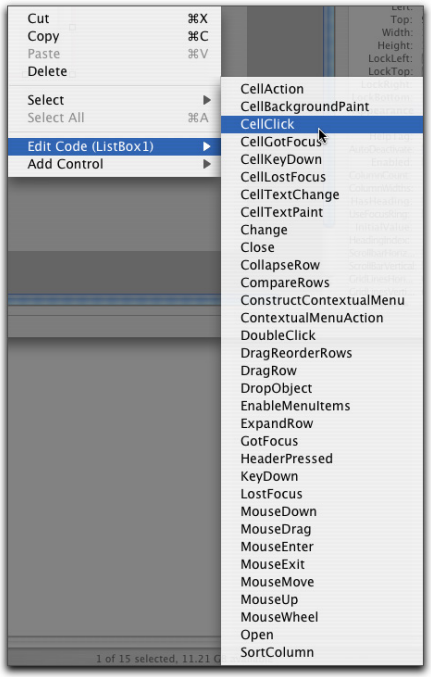
Each control has its own default event handler. For example, a PushButton's default event is the Action event, a ScrollBar's default event is the ValueChanged event, and a ListBox's default event is the Change event.

To open a selected event in the Code Editor for a specific control, do this:

1. **Right+click (Control-click on Macintosh) on a control in the window to display the contextual menu for the control.**
2. **Use the Edit Code submenu to choose the event handler that you wish to open.**

The Edit Code contextual menu item for an individual control shows the events for that control. Use it to go to a particular event handler directly.

Figure 178. The Edit Code submenu for a ListBox control.



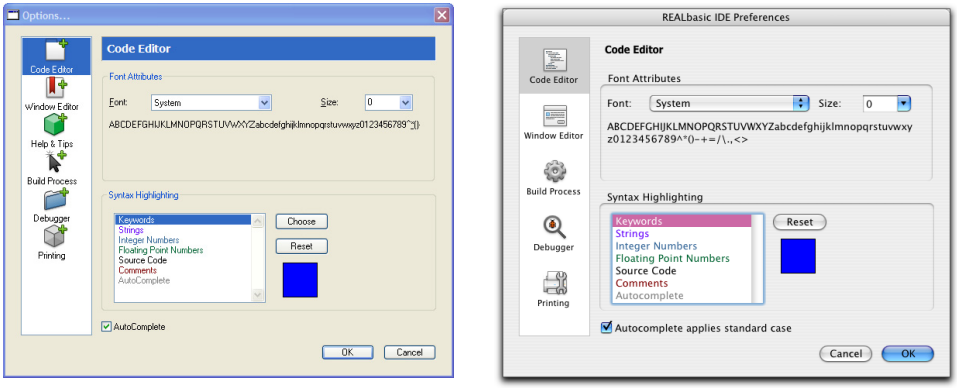
To open the Code Editor for a window, module or class, double-click the name of the item in the Project Editor or choose Edit Source Code from its contextual menu. Modules are stand-alone objects in which you can manage constants, methods, and properties. Classes are reusable objects that are based on existing objects in REALbasic. To open a window's Code Editor directly from the Project Editor, you can select it and press Option-Return on Macintosh.

You will learn more about modules in Chapter 6, "Adding Global Functionality with Modules" on page 297 and classes in Chapter 9, "Creating Reusable Objects with Classes" on page 419.

Configuring the Code Editor

You can specify various options for the Code Editor. On Windows and Linux, choose Edit ▶ Options to display the Options dialog box. On Mac OS X, choose REALbasic ▶ Preferences. The Code Editor Options is shown in Figure 179

Figure 179. The Code Editor Options screen.



With the Code Editor Options or Preferences screen, you can set preferences for the following items:

- **Font:** Controls the font and font size used for your code. The first two choices, System and SmallSystem, tell REALbasic to use the current system font for the platform on which REALbasic is running. Setting a font size of zero tells REALbasic to choose a font size that looks best for the current platform.
- **Syntax Highlighting:** This set of preferences sets the colors the Code Editor uses to make your code more readable.

  You can assign colors to the following items:

  - **Keywords:** REALbasic language elements, such as control structures (If, while, etc.), and data types (integer, color, double, etc.).
  - **Strings:** Literal strings used in code, such as literal text passed to the MsgBox function to display a message to the user.
  - **Integer Numbers:** Numbers with no decimal point.
  - **Floating Point Numbers:** Numbers that use a decimal point. The distinction between Number and Real Numbers here doesn't depend on the declared data type. For example, if you declare the variable i to be an Integer but set i=5.76, the "5.76" uses the color assigned to Floating Point Numbers.
  - **Source Code:** The text of your variables, controls, classes, operators, methods, properties, and so forth.
  - **Comments:** The text of comments inserted in your code. Comments are preceded by two slashes (//), the REM keyword, or a single quote mark (').
  - **Autocomplete:** The text that appears when you use the AutoComplete feature while writing code. For more information on how to use Autocomplete, see the section "Autocomplete" on page 232.

  To change a color, select the item that you want to modify in the Syntax Highlighting list and click on the color patch to display the Color Picker, select a new color,

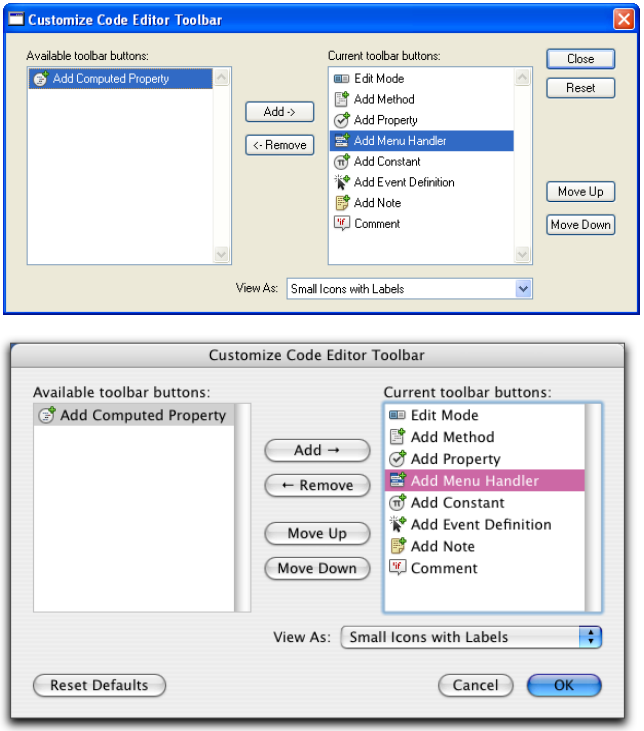
and click OK. To revert all color preferences to their default settings, click the Reset button.

- **Autocomplete applies standard case:** If you select this preference, REALbasic uses the standard uppercase/lowercase conventions when you use the Code Editor's autocomplete feature. It automatically capitalizes terms and uses the internal capitalization shown in the *Language Reference*. If this preference is deselected, autocomplete uses whatever capitalization you've typed. For more information on the Autocomplete feature, see the section "Autocomplete" on page 232.

Customizing the Code Editor Toolbar

The Code Editor Toolbar (just below Tab bar) has buttons for adding items to the code for the object being edited. By default, it has buttons for adding methods, properties, event definitions, constants, menu handlers, notes, and comments. If you like, you can modify the Code Editor toolbar with the View ▶ Editor Toolbar ▶ Customize submenu. Note that a Code Editor pane must be selected to customize the Code Editor toolbar rather than another editor's toolbar. The Customize Code Editor Toolbar dialog appears:

Figure 180. The Customize Code Editor Toolbar dialog box.



The Customize Code Editor Toolbar dialog box uses a "mover" interface to configure the toolbar. Listed in the right panel are the current items in the toolbar.

The left panel contains any available items, but all Code Editor toolbar items are displayed by default.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (assuming that an item is available).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down menu. Your choices are:
  - Big icons with labels.

  - Small icons with labels,

  - Big icons (no labels),

  - Small icons (no labels),

  - Labels only.

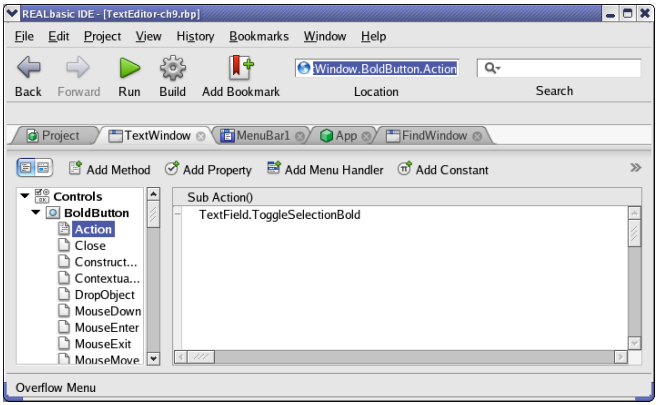
- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or Reset Defaults button (Macintosh).

The Browser

To view the items in each category, click the plus sign (Windows) or disclosure triangle (Macintosh and Linux) to the left of the category name. For example, to view all of the controls for the window, click the plus sign or disclosure triangle next to the Control's category name in the window's editor. When you do this, the list of controls will appear below and to the right. Each of the controls can then be expanded in the same way to display a list of the event handlers for that control.

For example, in Figure 181 you can see that the window named TextWindow has a BevelButton named BoldButton.

Figure 181. A BevelButton's Action event handler.



The Location area gives the name of the screen as TextWindow.BoldButton.Action. That is, it's the name of the window, followed by the name of the control, followed by the name of the event handler, separated by dots: *WindowName.ControlName.EventHandlerName*.

This control has the following event handlers:

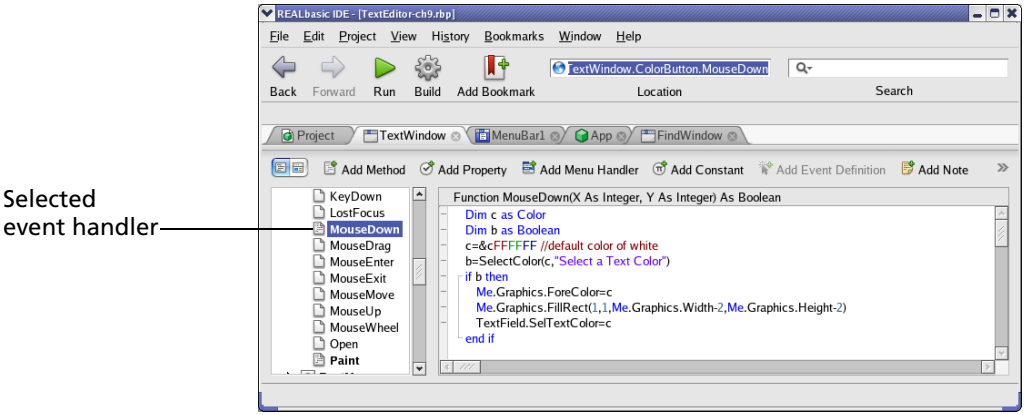
- Action
- Close
- DropObject
- MouseDown
- MouseEnter
- MouseExit
- MouseMove
- MouseUp
- Open

Event handlers are listed in alphabetical order within a control or object. When you first open the event handlers for a control (for example, by double-clicking a control in a window), its default event handler is selected. It may be the first event handler in the list (i.e., the Action event handler for a PushButton) but this is not always true. For example, the Paint event is the default event handler for a Canvas control.

Clicking on a control's event handler in the Browser list displays the code associated with that event handler in the Code Editor.

You will learn more about these event handlers later in this chapter.

Figure 182. Some code associated with a control's MouseDown event handler.



Event handlers in the Browser that have code associated with them appear in bold. If one of a control's event handlers has code in it, the event handler's name, the control's name, and the Controls category will all appear in bold. For example, in Figure 181, only the Action event has any code associated with it. Controls that have no code appear in plain text. When you are trying to find some code, the bold style acts as a visual cue to let you know if there is any code you might need to look at.

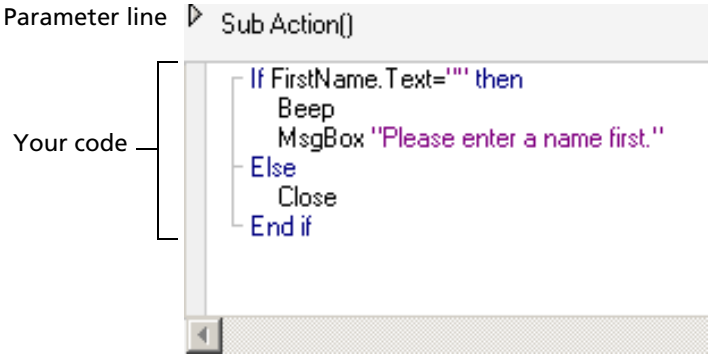
For more information on the Code Editor's contextual menu, see the section "The Code Editor's Contextual Menu" on page 237.

NOTE: When you add new controls to a window, REALbasic gives them default names. For example, the first PushButton you add to a window will be named "PushButton1" by default. A name like that describes the type of object but not what it does. The Browser displays small icons next to each control's name to make the control type clear. Its icon indicates the class from which the control was derived. You should use the control's Properties pane to give the control a meaningful name. For example, in Figure 181 on page 225 a BevelButton control has been renamed BoldButton to indicate its function in the application.

Understanding Methods in the Code Editor

Event handlers, menu handlers, and methods are all, in fact, methods. Event handlers and menu handlers are simply methods that are called when certain events occur or menu items are selected. When you select a method in the browser, its code appears in the Editor. Methods are made up of two parts: The parameter line and your lines of code.

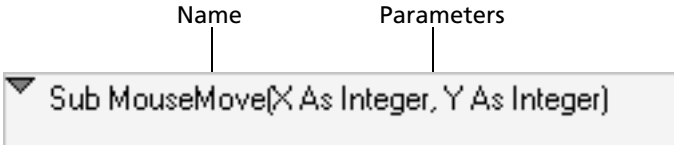
Figure 183. The parts of a method.



The Parameter Line

The parameter line displays Sub (short for subroutine) if the method does not return any values, followed by the name of the method or event handler, and then any parameters surrounded by parentheses. The example in Figure 184 shows the parameter line of a MouseMove event handler. This event handler is called whenever the mouse is moved inside the control. It is passed two parameters that describe the current mouse location. The parameter X represents the horizontal measurement and Y represents the vertical measurement. Your code for the MouseMove event handler can use the values of X and Y.

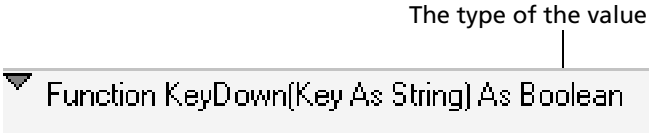
Figure 184. The parts of the parameter line.



For more information on parameter passing, see “Passing Values to Methods” on page 195 of chapter 4.

If the method returns a value, it’s called a *function*. A function’s parameter line begins with the word *Function* instead of *Sub* and has an additional parameter; the data type of the value that will be returned by the function. The declaration of the value returned by the function follows the parameters. Figure 185 shows the parameter line for an EditField’s KeyDown event handler. This event handler is called when the user types a key in an EditField. It is passed the key that was pressed in the parameter *key*. The value returned is a boolean. If you return True from the function, the event is discarded as if it never happened at all and the key that was pressed will not appear in the EditField.

Figure 185. The parameter line of a function



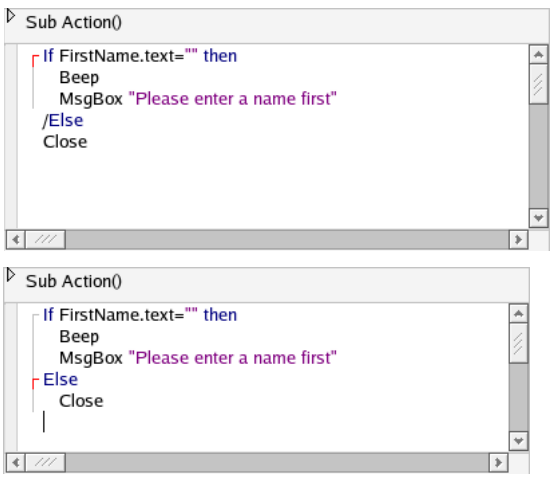
For more information on functions, see "Returning Values from Methods" on page 196 of chapter 4.

Entering Your Code in the Code Editor

As you enter your code, REALbasic does a few things for you automatically. First, it indents your If...Then, Select...Case, and loops as you type them to make it easier to see which lines of code fall inside a particular statement. It also indicates the scope of each level of indentation with gray brackets to the left of your code. For example, Figure 183 on page 227 shows how the lines indicate the scopes of the If and Else clauses in an If...Then statement. If the If...Then statement were incomplete, then the top corner of the bracket would be red rather than gray and the bottom section of the bracket would be missing.

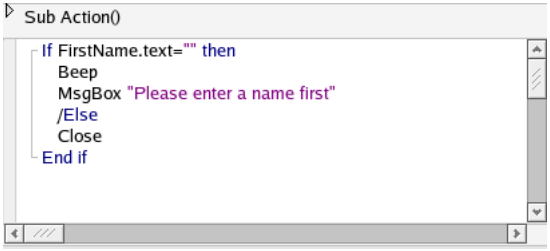
For example, in the top illustration in Figure 186, the Else statement is misspelled, so the bracket does not enclose the If...Else portion of the statement. The red corner indicates that the If statement is open. In the bottom illustration, the spelling error is corrected, but the If statement is still open because the End If statement hasn't been added.

Figure 186. Errors in If...Then statements.



In Figure 187, the If statement has been closed, but the Else statement is still misspelled. Notice that Else isn't indented as it should be and the bracket does not show that it is part of the If...Else...End if statement.

Figure 187. A closed If...End if statement with Else misspelled.

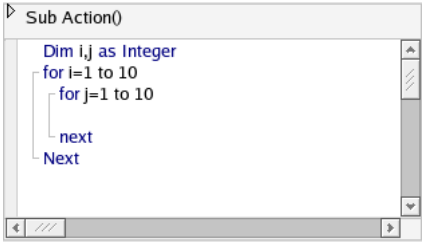


If your code uses two or more levels of nested If statements, the indentation and bracketing shows the scope of each statement. Errors are indicated in the same way as for single-level If statements.

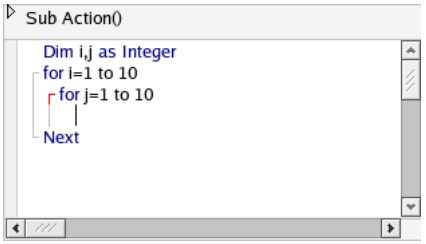
Figure 188 on page 230 shows a correct nested For...Next statement and two types of errors. In the middle illustration, the inner (nested) For...Next statement is incomplete, so its set of brackets has a red top corner and no bottom corner. In the bottom illustration, both loops are complete, but the outer loop is incorrect. It is ended by an (incorrect) "End For" statement, which doesn't exist in the language. The brackets for the outer loop are complete but both the top and bottom corners are red.

All of the errors that are indicated by incomplete or red brackets indicate errors that will prevent the application from being compiled. They must be fixed before you can test the application.

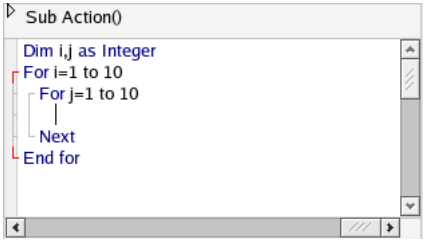
Figure 188. Nested For statements.



Correct: Each For...Next loop has its own brackets



Incorrect: The inner loop is incomplete, indicated by a red incomplete bracket

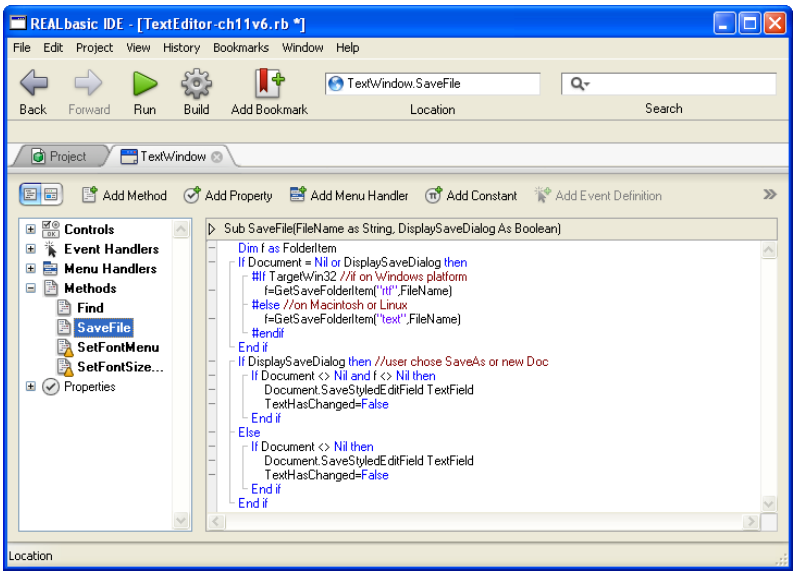


Incorrect: The outer loop is complete but incorrect. The "End For" must be replaced by "Next". Both top and bottom corners of the outer loop are red.

Second, it uses colors to indicate different types of keywords in your code. It uses blue text for keywords and data types (You can change this color by selecting another color using the Code Editor Options (Preferences on Mac OS X). See "Configuring the Code Editor" on page 221.) Figure 183 on page 227 shows an example of these features.

If you insert comments in you code, they appear in red type by default The color applied to comments can also be changed via Code Editor Options. A text string is treated as a non-executable comment if it is preceded by double slashes (//), a straight quote mark ('), or the REM keyword. Comments do not necessarily have to appear on separate lines, as is illustrated in Figure 189.

Figure 189. Comments at the end of lines of executable code.



A different color is also used for text strings that appear as part of executable code. This text is in purple, as shown in Figure 189.

Inserting a Color into the Code Editor

The REALbasic language includes three functions that return a color, RGB, HSV, and CMV. You can always use them to insert a color value into the Code Editor. You can also specify a color literal by writing its RGB value in hex and preceding it with the &c symbol, e.g. &cFF0000 specifies the color red. However, none of these ways display the color that you specify. If you want to choose a color from a color palette, you have the following option.

When you need to insert the value of a color into the Code Editor, you can take advantage of a shortcut that is in the Code Editor contextual menu. Place the insertion point where you want to insert the color value. Right+click (Control-click on Macintosh) and choose Insert Color from the contextual menu. The Color Picker for your operating system appears. Use it to select a color from its color palette and click OK. When you click OK, the value of the color in hexadecimal is inserted at the insertion point. It uses the format &cRRGGBB, where RR is the value of red in hex, GG is the value of green, and BB is the value of blue. Each ranges from 00 to FF.

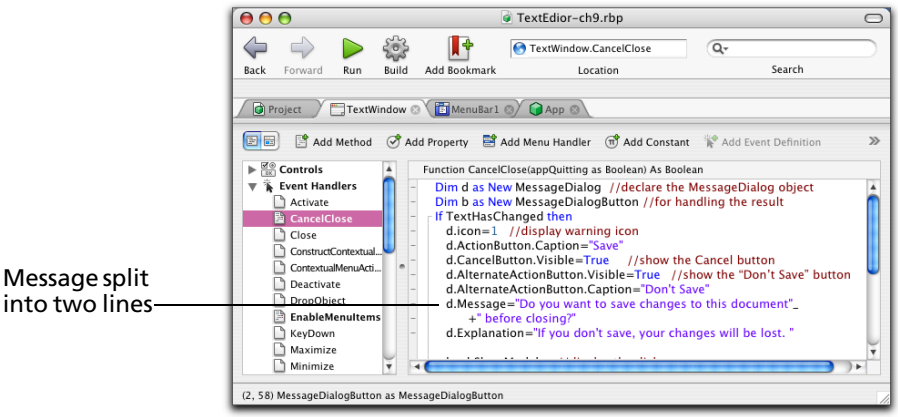
Breaking up a Line of Code in the Code Editor

If you are writing a very long line of code, you can continue it onto a second "line" in the Code Editor. End the first line with an underscore character and continue typing on the next line. The REALbasic compiler will treat the continuation line as part of the first line of code.

For example, if you need to assign a long string to a property or variable, you can split it up into two lines in the Code Editor. You do this by placing an underscore character as the last character on the line to be continued. When you use the underscore for this purpose, the continuation lines will be indented automatically, as shown in this following example. The text assigned to the Message property of a MessageBox is split into two lines in the Code Editor, but the compiler treats it all as one string.

```
d.Message="Do you want to save changes to this document"_
      +" before closing?"
```

Figure 190. Using the underscore character to continue a long line.



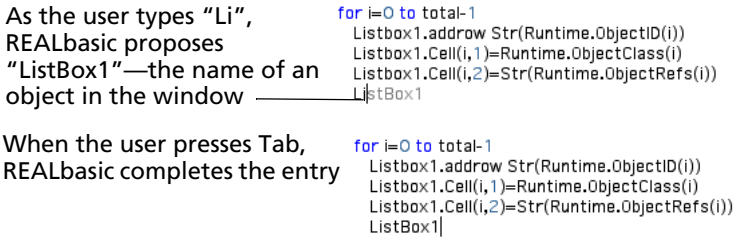
Notice that this example places the underscore character outside the literal text string and uses the + operator to append the second string to the first. You need to do this so that the compiler doesn't think that you are trying to include the underscore character as part of the literal text string. If the line that you are splitting up does not involve a quoted string, you don't have to do this.

To type the underscore, enter Ctrl+Enter (Command-Return on Macintosh). This enters the underscore and moves the text insertion point to the next line.

Autocomplete

As you type, REALbasic also attempts to guess what you are typing. If you type the first few characters of a REALbasic language object — either built in or a variable, method, or property that you created — it shows its guess in light gray type. If the guess is correct, complete the entry by pressing the Tab key. This process is illustrated in Figure 191 on page 233.

Figure 191. REALbasic proposes "ListBox1 when the user types "Li"



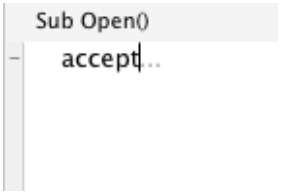
If REALbasic finds several matching objects, it instead displays an ellipsis ("…"). Press the Tab key to display a contextual menu of choices. You can select an item on the contextual menu with the mouse or navigate up or down in the menu with the Up and Down Arrows. When using the Down arrow keys and you reach the end of the list, the selection will wrap back to the start of the list (same if you use the Up arrow key and reach the top of the list). When the desired item is highlighted, you can select it by pressing the Spacebar, Return, Enter, or Right Arrow keys.

While the contextual menu is displayed, you can continue typing to narrow down the list of choices or cancel the contextual menu by pressing the Left Arrow, Esc, Delete, Backspace, or Clear keys.

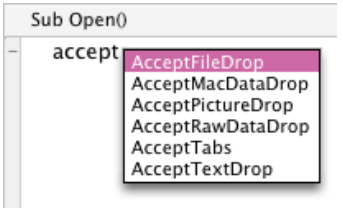
The process of choosing from the contextual menu is illustrated in Figure 192.

Figure 192. Multiple choice autocomplete options.

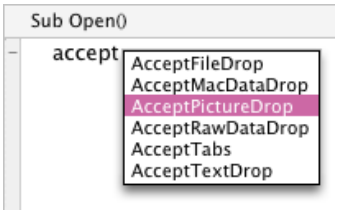
The user types "accept" and an ellipsis appears…



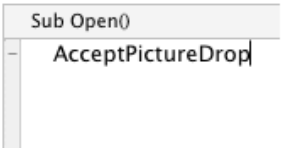
He presses Tab to display the contextual menu…



He uses the Up and Down Arrow keys to highlight the desired choice and presses Enter or clicks on it with the mouse.



REALbasic completes the entry.



Autocomplete works in the middle of an expression. With the insertion point anywhere in an expression, you can press Tab to see a pop-up menu of acceptable choices. Auto-code completion also works for user-defined properties, methods, functions, and events. Autocomplete works in the Location and Search areas in the Main toolbar as well.

It also works in the search window in the Online Reference.

Using the Edit Menu

While you are entering code, the Edit menu's standard Cut, Copy, and Paste commands are available. The Code Editor also supports both Undo and Redo (Shift+Ctrl+Z or Shift-⌘-Z). The Comment button or the Edit ▶ Comment menu command (Ctrl+' or ⌘-') is especially useful. When applied to lines of code, it comments them out; when applied to comments, the command changes to Uncomment and converts the lines back to code.

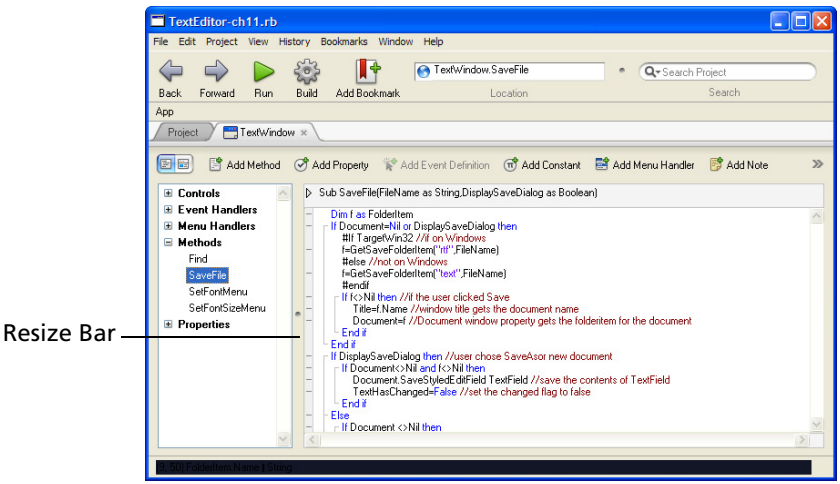
Note: The Comment button may not be shown; if you wish to use it, you can add it to the Code Editor toolbar by choosing View ▶ Editor Toolbar ▶ Customize or the Customize menu item in the toolbar's contextual menu and adding it to the toolbar.

Getting More Usable Space in the Code Editor

There may be times when you need more vertical or horizontal space in the Code Editor. You can, of course, resize the IDE window to get more space, but this isn't always an option. One way to get more space is to use a smaller font. You can set the font and font size for the Code Editor by choosing Edit ▶ Options (REALbasic ▶ Preferences on Mac OS X) and selecting the Code Editor font and font size in the Source Code Editor pane.

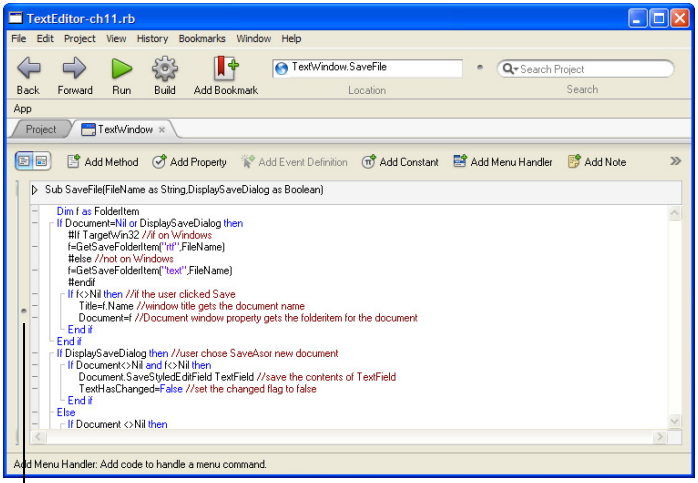
You can also minimize the Browser when you don't need it. Use the View ▶ Editor Only command to minimize the size of the browser area and maximize the code editing area. You can also hide the Browser by dragging the divider (the bar between the Browser and the Editor) all the way to the left side of the Code Editor window. However, dragging the divider to this position does not change the state of the Editor Only menu command. If Editor Only is off and you drag the divider to the extreme left, Editor Only is still off even though only the editor is visible.

Figure 193. The Code Editor's Resize Bar.



When you do this, the Browser is hidden and the divider is reduced to a button to the left of the Code Editor.

Figure 194. The Code Editor with the Browser hidden.



The Resize Bar handle

As you can see in Figure 194, this gives you quite a bit of horizontal space to work with in the Code Editor. You can show the Browser again by dragging the Resize Bar handle towards the right side of the Code Editor window.

NOTE: The Browser will expand and collapse the categories (Controls, Events, Menu Handlers, Properties) when they are selected by pressing Ctrl+Left Arrow (to collapse) or ⌘-Left Arrow and Ctrl+Right Arrow or ⌘-Right Arrow (to expand).

Opening a Window from its Code Editor

If you are working in a Code Editor that belongs to a window, you can switch to the window's Window Editor by pressing Ctrl+` (Command-` on Macintosh) or choosing the View ▶ Show Layout menu command. When the Window Editor is shown, the menu command changes to View ▶ Show Code. Pressing the keyboard shortcut will toggle back to the Code Editor for the window.

You can also switch views using the pair of icons just above the browser area. They toggle between the Window Editor and the Code Editor for the window. The icon on the left displays the Code Editor and the icon on the right displays the window in the Window Editor.

Figure 195. Buttons for switching between the Window Editor and the Code Editor.

You can also open up another IDE window for the project and display the Window Editor and the Code Editor views in separate windows. Choose File ▶ New Window to get a new IDE window for the current project. In the new window, you can use

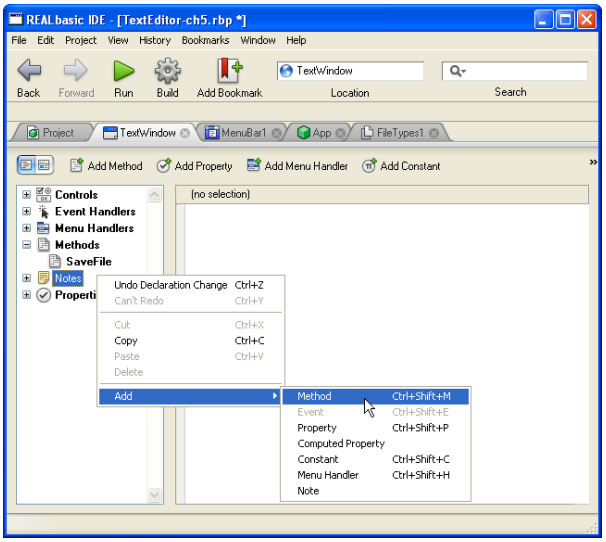
one IDE window to display the window in its Window Editor and the other IDE window to display its Code Editor.

The Code Editor's Contextual Menu

Contextual menus are context-sensitive pop-up menus that appear when you right+click the mouse button on an item (Ctrl+click within the Code Editor on Macintosh). This displays a contextual menu with all of the items from the Browser.

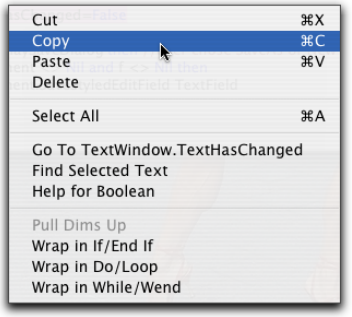
The Add menu has a submenu that allows you to add a new item to the Code Editor. For Code Editors belonging to windows, you can add a new method, property, constant, menu handler, or note. For Code Editors belonging to classes, you can also add a new event definition.

Figure 196. The Code Editor's Add contextual menu.



These menu commands are also available under the IDE's Edit and Project menus. The contextual menu in the code editing area has special items for searching on the current item and to wrap selected items in If, Do, and While statements.

Figure 197. The Code Editor's contextual menu in the code editing area.



Searching your Project

The REALbasic IDE offers three types of interfaces for finding project items. They are:

- The Search area in the Main toolbar,
- The Find and Replace dialog box,
- The Code Editor contextual menus.

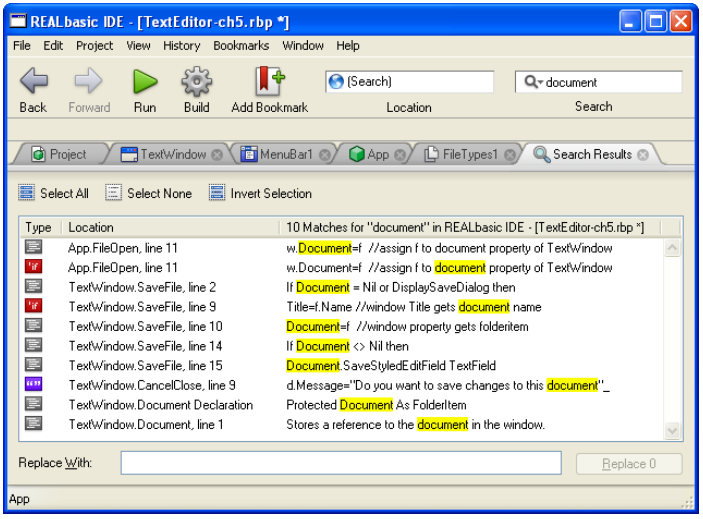
The Search Area

The Search area contains a menu in which you can specify that you want to search the entire project, the current item (for example, the current window, menubar, class, or module), or the current method only. On Mac OS X 10.4, it also offers to search the entire computer using the operating system's search engine, Spotlight. You display the pop-up menu by pressing the mouse button on the magnifying class in the Search area.

To do a search, choose the scope of the search from the pop-up menu and then type in the item to be searched. Press the Enter key (Return on Macintosh). When it finishes searching your project, it will display the results of the search in a new IDE screen called Search Results. If you do a new search, those results will replace the current results.

For example, Figure 198 shows a search on the string "Document". In the project, it is the name of a window's property. REALbasic has found all occurrences in the project and listed them on the Search Results page. The Type column in the list shows the type via an icon and the Location column gives the location in the form *Classname.EventHanlder* or *WindowName.EventHandler*. The Matches column gives the line of code that contains the term being searched.

Figure 198. The Search Results for a search on "Document".



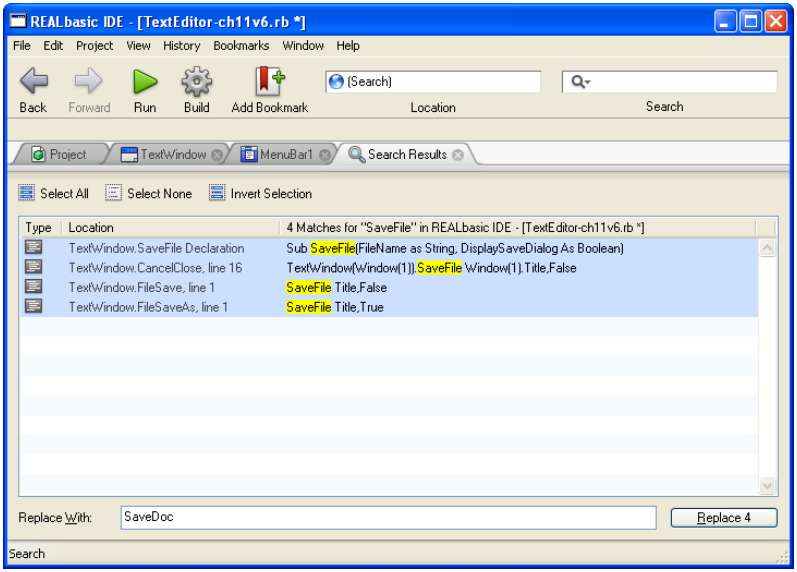
The columns in the Search Results list are sortable. Click on the column header to sort on that column.

In this search, the search string is the name of a property. The icon in the left column indicates whether its usage is in executable code, in a text string, or in a non-executable comment.

The Search Results screen gives you the option of replacing the highlighted string. If you want to do a replace, enter the replacement string into the Replace With area at the bottom of the window. Then highlight one or more lines in the list. You can use the Select All, Select None, and Invert Selection buttons in the Search Results toolbar to make or change the selection.

The replacement will be done only in a highlighted line. As you highlight the lines, the Replace button will indicate how many replacements will be done. For example, in Figure 198, no lines are highlighted, so no replacements will be done. In Figure 199 all the lines are highlighted so all the instances will be replaced.

Figure 199. The Search Results screen with all rows selected.



If you don't want to do a replace, you can navigate to any of the search results by double-clicking the line. The Search Results screen remains accessible via the Tab bar after you have visited the editor containing the searched string. When you are finished dealing with the occurrence of the search string, you can return to the Search Results screen by clicking on its tab.

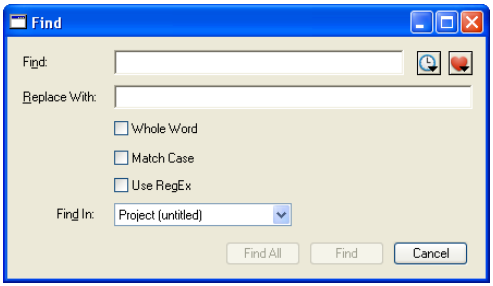
Find and Replace dialog

The Find/Replace dialog (Edit ▶ Find ▶ Find or ⌘-F or Ctrl+F) has several options that are not available in the Search area.

Like the Search area, the Find and Replace dialog gives you the option of searching the entire project, the current item, or the current method, if here is one.

If the you are displaying the Project Editor when you open the Find and Replace dialog, you can choose the current item by clicking on it in the Project Editor. It does not have to be open.

Figure 200. The Find/Replace dialog box.



You can choose to search in:

- **The entire Project:** Searches all of the project items in your Project Editor. This is not limited to classes and windows.
- **The current window or class and their subclasses:** Searches only the current window or class and all subclasses of that window or class.
- **The current project item only:** Searches only the current class, window, module, and omits any subclasses derived from the current class or window.
- **The current method only:** Searches only the current method in the Code Editor. Available only when a method is displayed in its Code Editor.

The CheckBoxes in the center of the Find dialog offer three options:

- **Whole Word:** Find will search only for the entire word or words specified in the Find area. It will ignore matching text strings that are embedded within words.
- **Match Case:** Find will search only for text strings that match the uppercase/lowercase specifications entered into the Find area. For example, if you entered “document”, and checked Match Case, it will not find “Document”.
- **Use RegEx:** If RegEx is specified, you can use regular expressions in your find and replace specifications. REALbasic will interpret them according to the rules for regular expression searches. If you use regular expressions but do not check this CheckBox, REALbasic will assume that you mean the literal text you entered. See the entry for the RegEx class in the *Language Reference* for information about regular expressions.

The Find/ submenu gives you the ability to find the next occurrence of the item you are searching for, replace the highlighted text in the Code Editor with the text in

the Find window's Replace field, and replace all occurrences within the chosen scope. The keyboard equivalents are shown in Table 8:

Table 8: Keyboard equivalents for Find commands.

Command	Keyboard Equivalent
Find	Ctrl+F or ⌘-F
Find Next	Ctrl+G or ⌘-G
Find Previous	Shift+Ctrl+G or Shift ⌘-G
Replace and Find Next	Ctrl+T or ⌘-T

Recent and Favorite Searches

The two icons in the top-right corner of the Find dialog box enable you to store and recall previous searches. The icon on the left represents recent searches and the icon on the right represents favorite searches. REALbasic adds your searches to the Recent searches list automatically and you can add searches to your Favorites list with its Add to Favorites menu item.

Figure 201. The Recent searches and Favorite searches icons.

To recall a recent search, simply display the Recent searches pop-up menu and choose it from the list. To add a search to your Favorites list, display the Find or Find and Replace specification in the Find dialog and then choose Add to Favorites from the Favorites pop-up menu. REALbasic will then present a dialog box enabling you to name the favorite. You can then recall the stored search at any time by choosing its name from the Favorites pop-up menu.

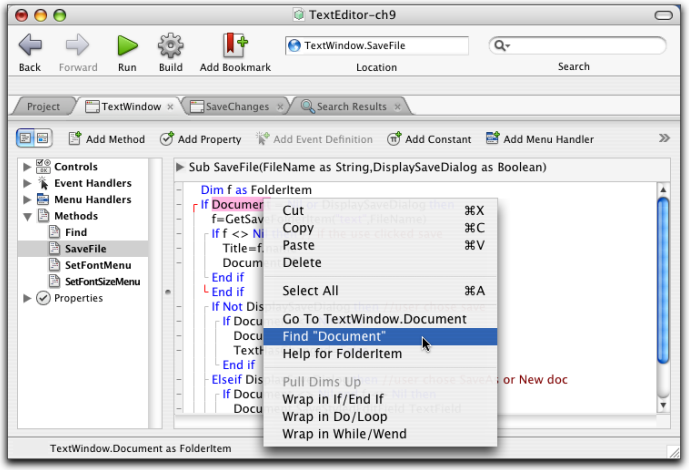
Finding using the Contextual Menu

If you are specifically looking for the occurrences of a variable, constant, property, method or other item, you can use the Code Editor's contextual menu to do the search. The "Find *Item*…" menu item finds the occurrences of *Item*. It is especially convenient for items other than local variables (which are declared with Dim statements within the method) because it searches through the code belonging to other objects in the project. For example, you can easily locate the declarations for the constants you use in localizing your application or properties that belong to other windows, modules, or classes. Or, you can use Find Item to locate the source code for one of your methods.

To use the contextual menu, simply select the item whose occurrences you wish to find. Right+click (Control-click on Macintosh) to display the contextual menu. Choose the "Find *itemName*" menu item. The Find contextual menu item is available in both the code editing area and the Code Editor browser.

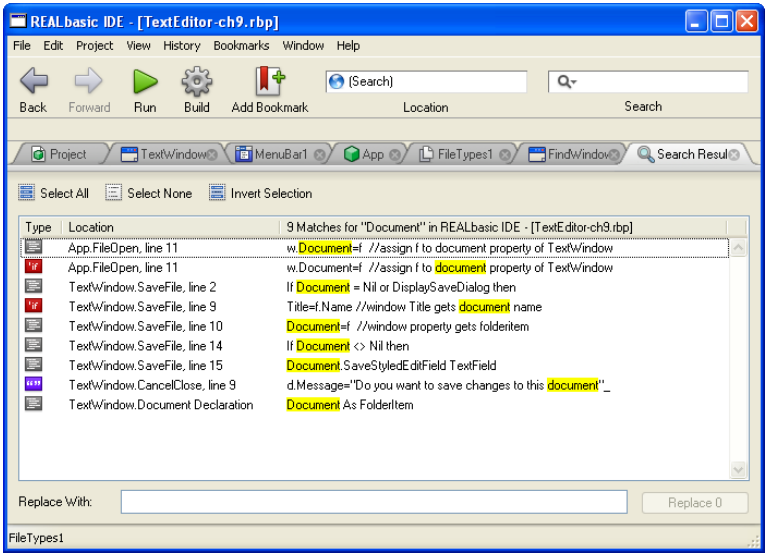
In the following example, the occurrences for of item "Document" in the Code Editor is located. It turns out to be a property of the window.

Figure 202. Using the "Find Item" contextual menu.



If the item occurs more than once, REALbasic opens a new search results screen and lists all occurrences, with the item being searched for highlighted. Figure 203 shows the results of the search that is specified in Figure 202.

Figure 203. The search results screen.



The standard Search Results screen offers the option of replacing the selected text in any number of lines. Also, you can double-click any row of the table to display that occurrence.

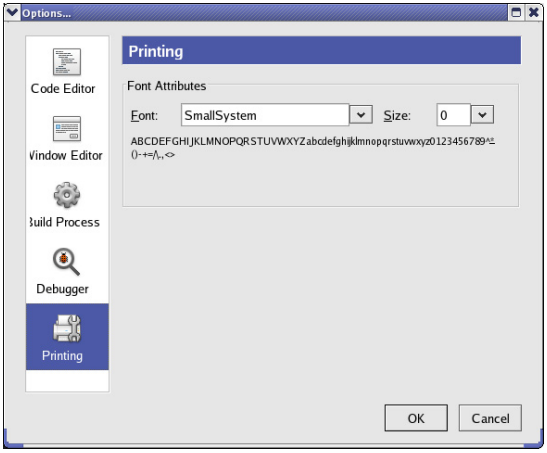
Printing Your Code

When you need to print your source code, choose File ▶ Print (⌘-P or Ctrl+P).

Using Edit ▶ Options (REALbasic ▶ Preferences on Mac OS X), you can specify the font and font size used for printing, as well as whether you wish to see keywords in bold, and print the colors that appear in the Code Editor. Use the System or SmallSystem fonts to tell REALbasic to use the system font for the platform on which REALbasic is running and a font size of zero to tell REALbasic to choose the optimal font size for your operating system.

The Printing options shown in Figure 204.

Figure 204. Options for printing.



Importing and Exporting Your Classes, Menus, Modules, and Windows

REALbasic makes it easy to import and export the various objects you can create. You can also import files you wish to use in your project, such as REALbasic code, windows, menus, sounds, pictures, QuickTime movies, REAL SQL databases, and resources.

External Project Items

If you want to share an object among several projects, you can import it as an external project item. The item remains on disk and changes that are made to it from another project are automatically reflected in the current project when you re-open it. However, you can't open more than one project simultaneously that references the same external project item. To add an external project item to a project, hold down the Ctrl+Shift keys (⌘ and Option keys on Macintosh) while you drag the item from the desktop to the Project Editor. In the Project Editor, the external project item's name will be shown in italics.

For more information, see the section "External Project Items" on page 56.

Importing

To import a file you wish to use in your project, simply drag it from the desktop and drop it in your Project Editor. Or, if the file is not conveniently located on the

desktop, choose File ▶ Import. An open-file dialog box appears, allowing you to navigate to and import the file.

For code, open the method that you wish to import code into and drag the text clipping into the body of the method. You cannot use the File ▶ Import command to import text files into the body of a method.

To delete a file that has been added to the Project Editor, highlight it and press the Delete key on the keyboard or choose Edit ▶ Delete. You can also delete an item in the Project Editor by Control-clicking on the item and choosing Delete from the contextual menu.

Note: When you import a REALbasic object by dragging it into the Project Editor, it automatically replaces an item with the same name, without asking you whether you want to replace it with the dragged item.

Some of the items you import are copied into your project. Some types of objects are not copied but instead an alias to the original file is stored inside your project. In the Project Editor, aliases are shown in italics.

When you build a stand-alone version of your project, most of these files are then copied into the stand-alone application. Table 9 on page 244 shows how all of the different file types are handled.

Table 9: How REALbasic handles imported files.

File Type	Copied Into Project?	Copied into stand alone applications?
Bitmap, PICT, JPEG, GIF	N	Y
Cursors	Y	Y
PowerPC Shared Libraries	N	N
QuickTime Movies	N	Y
REAL SQL Databases[a]	N	N
REALbasic Classes	Y	Y
REALbasic Menubars	Y	Y
REALbasic Modules	Y	Y
REALbasic Windows	Y	Y
Resources	N	Y
Sounds	N	Y
AppleScripts	N	Y

a. Data sources, such as a REAL SQL database, appear in plain text in the Project Editor even though the data is stored outside the project. The connection information to the data source is stored within the project.

Because REALbasic stores aliases to your imported files, they can be renamed and even moved. If both the project file and the imported files are moved to another

drive, REALbasic may have trouble locating the files. Should this happen, REALbasic will ask you to locate any files it can't find.

When you import a picture into your project by dragging it into the Project Editor or choosing Import from the File menu, the picture will be loaded into memory when you run your application regardless of whether or not the picture is used.

If you have several large pictures, your application may not behave very well or it could run out of memory. If you have large pictures, consider storing them externally and loading them with the GetFolderItem method.

All file types except REAL SQL databases are included in the stand-alone version of your application, so there is no need to include them with your application when you distribute it.

Exporting

The code for methods, events, constants, properties, and so forth can be dragged out of the Code Editor to the desktop as text clippings or into a text or word processor (Macintosh only). On Windows and Linux, you can copy code to the Clipboard and paste into a text editor or word processor.

You can also export your source code to a text file or in REALbasic's native format using an Export… menu command. Which method you use depends on what you will be doing with the exported code. If you are going to be including code in some kind of documentation, drag the code to the other application or export your code to a text file.

On the desktop, the exported objects have their own REALbasic icons, shown in Figure 205.

Figure 205. Icons for exported REALbasic objects.

You can export REALbasic objects using the Export command.

To export using the Export… command, do this:

1. **Open the item so that it is displayed on the screen or select it in the Project Editor.**
2. **Choose File ▸ Export Window/Menu/Module/Class or display the item's contextual menu and choose Export.**
3. **When the Save As dialog box appears, type a name and click the Save button.**

   On Windows and Macintosh, the Save As dialog gives you the option of saving as a REALbasic object or as XML.

Encrypting Your Source Code

If you want to distribute a copy of a window, menu bar, module, or class for others to use but you do not want them to be able to view or edit your code, use the Encrypt command to protect the object prior to exporting it. The icon belonging to an encrypted item in the Project Editor has a small key in its lower-right corner.

When a protected object is exported, its icon is no different from an unprotected object.

You can encrypt (protect) or decrypt (unprotect) a window while it is in your project. An encrypted window cannot be opened in a Window Editor and no one can access any code associated with the window or any of the controls on the window.

When encrypting an item, you supply a password that can be used to decrypt it later.

To encrypt an item, do this:

1 **Right+click (or Control-click on Macintosh) on the item in the Project Editor and choose Encrypt from the contextual menu or choose Edit ▸ Encrypt.**

You can optionally add an Encrypt button to the Project Editor toolbar. If you do so, you can highlight an item in the Project pane and click the Encrypt button to encrypt the selected item.
The Encrypt dialog box appears, as shown in Figure 206.

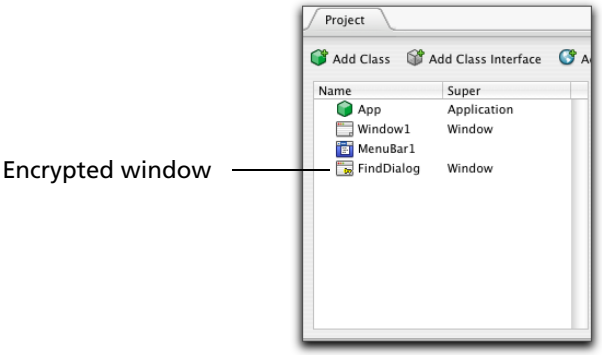
Figure 206. The Encrypt Dialog box.

2 **Enter and confirm a password for decrypting and click the Encrypt button.**

Important Note: Don't forget the password.

An encrypted item appears in the Project Editor with a small key in the lower right corner of its icon, such as this icon for the encrypted SaveChanges window.

Figure 207. A project with an encrypted window.



When a programmer (including the original author) tries to open an encrypted item, the Decrypt Object dialog box appears, shown in Figure 208.

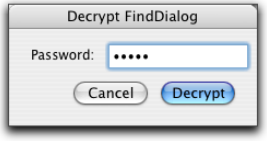
To edit the object and/or its code, you must decrypt it using its encryption password.

To decrypt an encrypted object, do this:

1 **Right+click (or control-click on Macintosh) on the item in the Project Editor and choose Decrypt from the contextual menu or choose Edit ▸ Decrypt.**

The Decrypt *Object* dialog box appears.

Figure 208. The Decrypt window dialog box.



2 **Enter the password that was entered when the item was encrypted and click Decrypt.**

If the password is correct, the key will disappear from the item's icon in the Project Editor, indicating that it has been successfully decrypted. If you entered an incorrect password, a dialog box will inform you of that fact.

If you have exported an encrypted item and then imported it into another project, you can decrypt it with the original password.

Responding To User Actions with Event Handlers

The applications you create with REALbasic are event-driven. This means that the user takes some action which results in something happening. For example, the user chooses Print from the File menu to print something or clicks a button to confirm a message in a dialog box. The user takes an action, and the application reacts to that action. The user's actions are called *events*. Earlier in this chapter, you learned that some events are caused directly by the user. For example, the Action event of a

PushButton occurs when the user clicks the PushButton. Other events are indirectly caused by the user, such as the Open event of a window that occurs when the window opens.

The key to writing the code for your applications is to know what events (both direct and indirect) you can respond to.

Object-Oriented Programming

REALbasic's programming language is *object-oriented*. This means that the code that is executed in response to an event is actually part of the object itself. Code that handles an event is called (appropriately) an *event handler*.

Objects can have their own methods. This allows you to associate code with an object even though it may not be executed in response to an event directed at that object. For example, suppose you have a window that displays the contents of a document and allows the user to edit it. It would make sense that the window would know how to save changes made to the document. You can add a method to the window that is called automatically when the user indicates that he wants to save changes to the document.

Because objects in your application are supposed to be just like objects in the real world, you want to associate code with the object that it truly belongs to. For example, if you want a window to change its size automatically when it opens based on certain conditions, it makes the most sense to put that code in the window's Open event handler. On the other hand, if you want a button to be enabled or disabled when the window opens that the button is a part of, you would put that code in the PushButton's Open event handler because the code affects the button. The code works perfectly in both places, but it is more object-oriented to associate it with the PushButton, since it affects the PushButton. For example, in the real world, when the door to the room you are in suddenly opens, you probably turn to look at it to see why it opened. The door does not turn your head. You have that ability to react to the door opening (an event). You choose to handle that event by turning and looking in the direction of the door. That ability is part of you — just as the code to enable or disable the button when the window opens should be part of the button and not the window.

Another benefit of associating code with the appropriate object is that the code goes with the object when you use the object elsewhere. If the code is not associated with the object, you will have to look for it or rewrite it. When you go somewhere, you take your computer skills with you because they are part of you.

Windows Events

Windows get many different events. Table 10 describes these events in general. If you need specific information about window events, see the Window class in the *Language Reference*.

Table 10: Window class events.

Event	Description
Activate	The window is being made the active window of the application.
CancelClose	The Quit method has been called so the application is about to quit. Returning True from this method will cancel the quit and the application will remain open.
Close	The window is about to close but hasn't closed yet. Controls also receive Close events. A window receives its Close event after all of the controls have received their Close events.
ConstructContextualMenu	Occurs when it is appropriate to display a contextual menu. Build and display the contextual menu in this event for the window or the ConstructContextualMenu event for a RectControl (any visible control).
ContextualMenuAction	The user has chosen an item from the contextual menu but it has not been handled by the menu item's Action event or its menu handler. Handle a contextual menu selection from a contextual menu created and displayed in the ConstructContextualMenu event handler.
Deactivate	The window is being deactivated.
DropObject	A file, piece of text, or a picture has been dropped on the window itself (not on a control in the window). This event handler is passed a parameter that gives you access to the item dropped.
EnableMenuItems	While the window is front of all other windows, the user has clicked in the menu bar to select a menu item or pressed a menu item's keyboard equivalent. This event handler gives you a place to decide which menu items should be enabled before the user can actually choose one.
KeyDown	A key has been pressed that has to be handled by the window. For example, the tab key is never sent to any control. It is instead handled by the window itself. If the window has no controls that can receive the focus, any keys that are pressed will generate KeyDown events for the window. This event handler is passed a parameter that tells you which key was pressed.
MouseDown	The mouse button has been pressed and has not yet been released. You can return False in this event handler to filter the event causing the window to act as if the mouse button was never clicked. This event handler receives parameters that indicate where the mouse was clicked in local window coordinates.

Table 10: Window class events. (Continued)

Event	Description
MouseDrag	The user has moved the mouse inside the window (but not over a control) while the mouse button is held down. This event handler receives parameters that indicate where the mouse is in local window coordinates.
MouseEnter	The user has moved the mouse inside the window from a location outside the window.
MouseExit	The user have moved the mouse outside the window from a location inside the window.
MouseMove	The user has moved the mouse inside the window. This event handler receives parameters that indicate where the mouse is in local window coordinates.
MouseUp	The mouse button has been released inside the window. This event will not occur unless you return True in the MouseDown event handler. The idea behind this is that if the mouse was never down, it can't be up. This event handler receives parameters that indicate where the mouse was released in local window coordinates.
Moved	The window has been moved by the user or by code that changes the window's Left or Top properties.
Open	The window is about to open but hasn't been displayed yet. Controls also receive Open events. A window receives its Open event after all of the controls have received their Open events.
Paint	Some portion of the window needs to be redrawn either because the window is opening or it's been exposed when a window in front of it was moved or closed. This event handler receives a Graphics object as a parameter which represents the graphics that will be drawn in the window. Graphics objects have their own methods for drawing graphics. See the Graphics class in the *Language Reference* for more information.
Resized	The window has been resized by the user or by code that changes the window's Width or Height properties.
Resizing	The user is in the process of resizing the window.
Restore	The window is being restored from its minimized state to its state prior to being minimized.

Opening Windows

There are two different techniques you can use to open windows. The technique you use depends on what you are going to do with the window once it's open. If your application will never have more than one copy of a particular window open at a

time, you can open the window simply by making reference to any of the window's properties or by using the window's Show method.

The following example opens a window by accessing one of the window's properties, its Title property:

```
aboutBoxWindow.Title="About My Application"
```

If you don't need to change any properties of the window, you can simply call its Show method to open it, as in this example:

```
aboutBoxWindow.Show
```

This technique works when you will only have one copy of the window open at a time because the name of the window acts as a reference to the window. If you have two copies of the window open, REALbasic will access the window that is already open rather than opening a second copy of the window.

If your application may have more than one copy of a window open at a time, you need to use the New operator to explicitly create a new instance of the window. To use the New operator, you must have a local variable or a property defined as the window you are going to open. This variable or property is used to store a reference to the window once it has been created. You can then use this reference to access the window.

```
Dim w as aboutBoxWindow
w=New aboutBoxWindow
```

A shorthand way of accomplishing the same thing is to create and instantiate the reference to the window in the Dim statement. Do this by using the New operator as a modifier in the declaration:

```
Dim w as New aboutBoxWindow
```

This single line of code opens the aboutBoxWindow.

Because aboutBoxWindow is an object of type Window, you can also Dim the variable as a Window, as in this example:

```
Dim w as Window
w=New aboutBoxWindow
```

This is beneficial when your code may open many different windows and you can't be sure which window it will need to open, as in this example:

```
Dim w as Window
If theAltKeyIsDown then
 w=New secretAboutBoxWindow
Else
 w=New aboutBoxWindow
End if
```

You could, of course, dimension two different variables; one as secretAboutBoxWindow and the other as aboutBoxWindow. But that might be a bit more confusing, especially if you had ten possible windows.

Because windows are objects, you can also dimension the variable as an object, as in this example:

```
Dim w as Object
w=New aboutBoxWindow
```

There is less of a need to dimension a window variable as type Object than there is to use type "Window." However, you might use this technique when you are creating new instances of controls on the fly. With controls, you can have a variable storing a reference to many different kinds of controls. See "Creating New Instances of Controls On The Fly" on page 278 for more information. See "Accessing Items of Other Windows" on page 275 for more information on how to use window references.

Adding Properties to Windows

The properties of an object are pieces of information that help define the object. They're variables that are accessible in more than one event handler or method. Windows have many pre-defined properties such as their title, width, height, etc. You can also add your own properties to windows that allow you to store information that is specific to the instance of the window. Properties work like variables except that they belong to the window and can be accessed by any of the window's event handlers and any of the window's controls. You also have the option of making a property of a window accessible to code outside the window. In contrast, a variable that is defined by a Dim statement inside an event handler is local. It cannot be accessed outside that event handler.

For example, if you have a window that displays the contents of a document, you might need to keep track of whether the user has modified the data to determine if he should be given a chance to save changes when he quits your application. Where do you keep track of this? Since the window is effectively a representation of the document, you can add a boolean property called *Changed* to the window. When the user makes a change in the window that affects the document, your code can change

the value of the *Changed* property from False to True. Later, when the user closes the window, the code in the window's Close event handler can check the Changed property to determine if the user needs to be given the opportunity to save his changes. The syntax for accessing the properties you add to windows is the same as the syntax you use to access a window's pre-defined properties. For example to set the Changed property of a window called "myDocumentWindow" to True, you use the following syntax:

```
myDocumentWindow.Changed=True
```

The *Changed* property should not be changed (no pun intended) from anywhere but the window. It wouldn't make sense for another window to be changing this property. However, six months after you add a property to a window, you might have forgotten this fact and add some code to another window that changes the *Changed* property. To avoid this problem, you can make the *Changed* property *Protected*. Protected properties can be accessed only by the window they are a part of.

The Scope of a Property

When you define a property of a window, you must decide whether the property will be accessible only to code belonging to the window and its controls or to other event handlers and methods outside the window. Your choices are:

- **Public: Accessible from Anywhere:** A Public property can be read or set elsewhere in the project, not just within the window that "owns" the property. A Public property can be accessed by event handlers and methods outside the window using the "dot" notation: *windowName.propertyName*. Within the window's own event handlers and methods, you can access the property by its name.
- **Protected: Accessible from the current window and its subclasses:** A Protected property can be read or set only by the event handlers and methods of the window and the event handlers of the controls in the window. A Protected property is accessed by its name. Code outside the window that owns the property cannot "see" the property. If any code outside the window tries to access a Protected property, REALbasic will display an informative error message.
- **Private: Accessible from the current window only:** A Private property behaves like a Protected property but it cannot be accessed by other windows that are subclassed from the current window. A window that is subclassed from another window inherits its Public and Protected properties. For information about creating subclasses, see the section "Understanding Subclasses" on page 421.

Declaring an Array as a Property

A property can be an array. For example, if you want to declare a four-element integer array of properties called WindowParameters, you would write:

```
WindowParameters(3) as Integer
```

in the Declaration area. You can also declare a property as an array with no elements and add elements later. In that case, you would declare WindowParameters using empty parentheses:

```
WindowParameters() as Integer
```

This denotes that WindowParameters is being declared as an array. You can use the Append method to add elements or the Array method to convert a list of items into an array to add elements.

Setting a Property in the Properties Window

In some cases, you will want to be able to set the value of the property with the Properties pane, just as if it were a built-in property that is displayed in the Properties pane in the window's Window Editor. By default, this option is turned off; the new property can be set only via code. Also, some properties cannot be set in the IDE.

If you wish to display the property in the window's Window Editor, you need to set the "Show in Properties Window" option. You can set this option after you declare the property. If you give the new property a default value, the default value will be shown in the Properties Window.

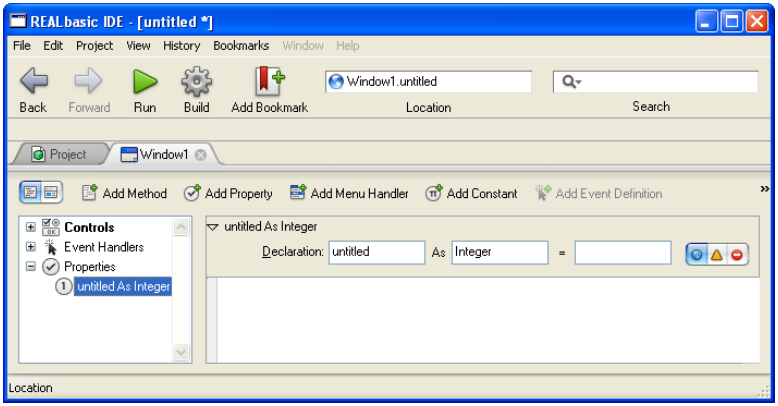
To add a property to a window, do this:

1. **Display the Code Editor for the window by clicking on its tab or, if it has no tab, double-click its name in the Project Editor.**
2. **If necessary, click the Code Editor button in the window's Editor Toolbar to access the Code Editor.**

Figure 209. The Code Editor button in the Window Editor toolbar.

3. **Click the Add Property button or choose Project ▸ Add ▸ Property.**
   A Property declaration area opens just above the code editing area.

Figure 210. The Property Declaration area.



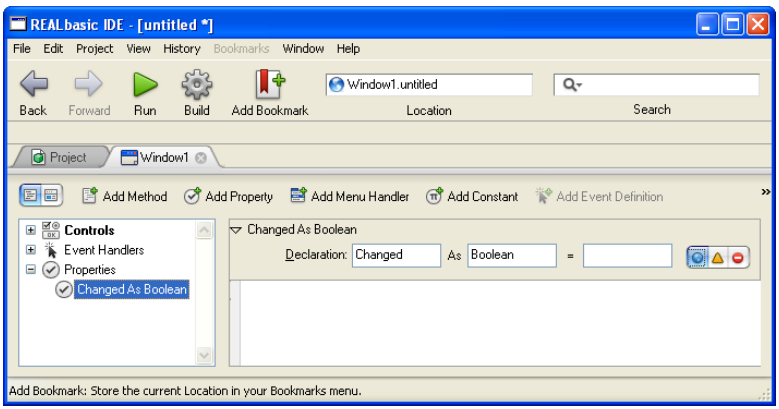
The Property Declaration area has three fields. They are for the name of the property, its data type, and its default value. The first two are required. If you do not provide a default value, the new property will take the default value for the data type that you choose. Strings have a default value of an empty string, numbers have a default value of zero, booleans have a default value of False, colors have a default value of black, and objects have a default value of Nil.

4 Fill in the Name and Data Type fields and, if desired, provide a default value.

For example, the Changed property would be entered with the name **Changed** and a data type of **Boolean**.

An example of a property declaration is shown in Figure 211.

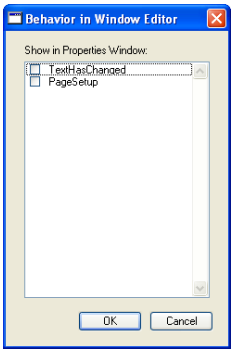
Figure 211. A completed property declaration.



5 (Optional) If you want the property to be listed in the Properties pane in the Window Editor, right+click (Control-click on Macintosh) the name of any property in the browser and choose Window Editor Behavior.

The Behavior in Window Editor dialog box appears. it lists all the properties that you have declared for the window that can be set in the IDE.

Figure 212. The Behavior in Window Editor dialog box.



This dialog box lists all the properties that are eligible to be added to the window's Properties pane.

6 Check each property that you want to have listed in the Properties pane.

The "Show in Properties Window" option means that, if you drag an instance onto a window, the property can be assigned a value from the Properties pane (rather than only with code).

7 Choose a Scope for the property by clicking on one of the three Scope buttons.

Your choices, from left to right, are Public, Protected, and Private. See the section "The Scope of a Property" on page 253 for information on Scope.

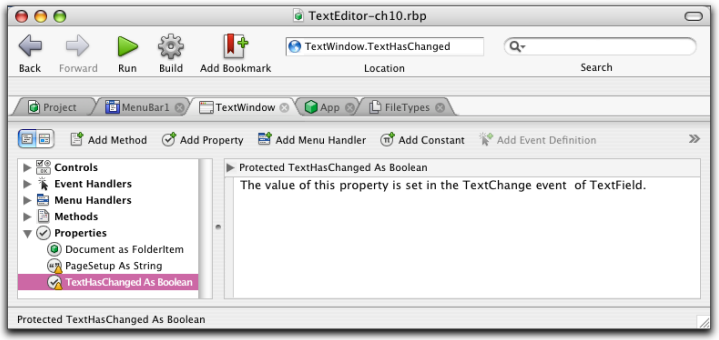
Figure 213. The Scope buttons.

8 (Optional) In the Code Editor area, add notes and comments about the property.

The text entered into the Code Editor for a property is automatically non-executable, even if you write valid REALbasic code. Add any comments you wish, including code samples.

You do not assign a value to the property in its Code Editor area. It is accessible from the event handlers and methods of the window, the window's controls, and, if the property is Public, event handlers and methods outside this window.

Figure 214. Property notes and comments in the Code Editor area.



Any text that you enter in the Code Editor for a property is not executable, even if it is code. When you add a comment to a property, the property declaration in the Browser area changes to boldface.

If a property is Protected or Private, the name of the property is preceded by either "Protected" or "Private."

To Edit a property you've added to a window, do this:

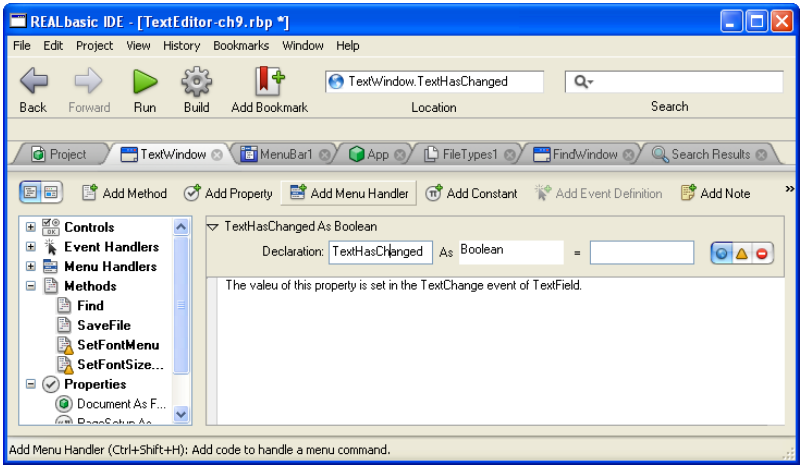
1 **Display the Code Editor for the window that contains the property.**

2 **In the Browser, expand the Properties category to display the list of properties for the window.**

3 **Click on the property to display the Property Declaration pane above the Code Editor.**

4 **If necessary, expand the Property Declaration pane by clicking its disclosure triangle (to the left of the property's name).**

Figure 215. The Edit Property pane.



5 **Make any necessary changes to the declaration.**

To delete a property from a window, do this:

1. **Open the Code Editor for the window that contains the property.**
2. **In the Browser, expand the Properties category to display the list of properties for the window.**
3. **Click on the property you want to delete to select it.**
4. **Choose Edit ▶ Delete or right+click (Control-click on Macintosh) and choose Delete from the contextual menu.**

The properties of a window can be accessed from any code within the window itself or any of its controls using the property name alone. The window name is not required as in this example that changes the window's title:

```
Title="My New Window"
```

In the absence of the window name, the current window is assumed. If you wish, you can always use the built-in function Self to indicate that you are referring to a window's property. When used in a control's event handler, Self refers to the parent object, which would be the window in which the control is located. In other words, the statement:

```
Self.Title="My New Window"
```

would also refer to the parent window's Title property.

Computed Properties

A computed property is actually made up of a pair of methods called Get and Set. Obviously this blurs the distinction between properties and methods. The Set method sets the value of a property (writes) and Get reads a value. You can implement either or both, making the computed property Read Only, Write Only, or Read/Write.

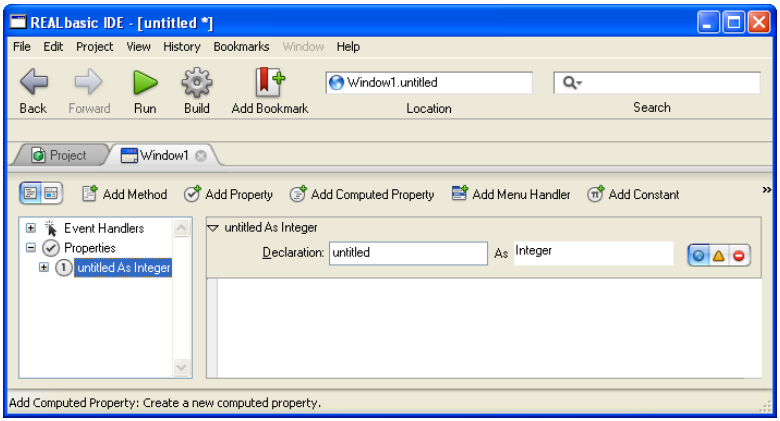
Computed properties can be delcared as shared. See the following section, "Shared Methods and Properties" on page 259 for information on shared properties and methods.

To create a computed property, do this:

1. **Click the Add Computed Property button in the Code Editor toolbar or choose Project ▶ Add ▶ Computed Property.**

REALbasic adds an untitled property to the window and displays a Property Declaration area above the Code Editor.

Figure 216. The Declaration Area for a Computed Property.



2 **Enter the Name and Data Type for the computed property and set its scope.**

Notice that the browser area shows that you can expand the computed property.

3 **Click the plus sign (Windows) or the disclosure triangle (Macintosh and Linux) to reveal the Get and Set methods that belong to the computed property.**

The Get method returns a value of the declared data type. The Set method is passed the value of the property. You do not have to implement both methods. You can make the computed property Read Only, Write Only, or Read/Write.

4 **Write either the Get or the Set method or implement both methods.**

Shared Methods and Properties

A shared method or property is like a 'regular' method or property, except it belongs to the class, not an instance of the class. A shared method or property can be accessed without instantiating an instance of the class or from any instance of the class.

In contrast, "regular" methods or properties are considered *instance* methods and properties. This means that they belong to a particular instance of the class.

The Self keyword is not available in a shared method or shared computed property and you cannot access instance methods or instance properties inside a shared method unless you are doing so via an instance.

To create a shared property or method, do this:

1 **Pull down the Project ▸ Add submenu and choose Shared Method, Shared Property, or Shared Computed Property.**

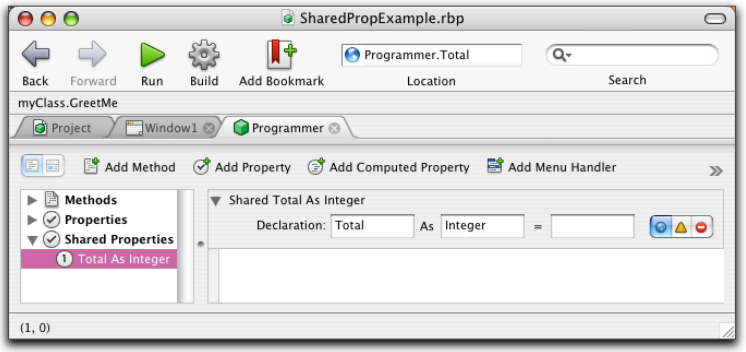
REALbasic displays the declaration area for the type of item you chose. If the category of item does not already exist, it is added to the Code Editor browser area.

2 **Declare the property or method and set its scope the normal way.**

3 If it is a method, write the method in the Code Editor.

Figure 217 illustrates a shared property in the Code Editor.

Figure 217. A Shared property in the Code Editor.



Here is an example that illustrates the difference between instance and shared methods. Create a class, myClass, in the Project Window and create the following simple method of myClass:

```
Sub WelcomeMe(a As String)
  MsgBox a
```

If you create WelcomeMe as an instance method, you can only call it from an instance of myClass, e.g.

```
Dim greetMe as myClass
greetMe = New myClass
greetMe.WelcomeMe "Hello World"
```

If you create WelcomMe as a shared method, you can call WelcomeMe with the line:

```
myClass.WelcomeMe "Hello World"
```

You can also call a shared method or property from any instance of the class, just like an instance method or class. The key advantage of shared properties and methods is that you can share them among all the instances of the class. For example, if you are using an instance of a class to keep track of items (e.g., persons, merchandise, sales transactions) you can use a shared property as a counter. Each time you create an instance of the class, you can increment the value of the shared property in its constructor and decrement it in its destructor. When you access it, it will give you the current number of instances of the class.

For example, consider the example in the section "Using Classes in Your Projects" on page 450. The local variable "person" stores a reference to the instance of the custom class "Programmer".

```
Dim person as Programmer
person=New Programmer
person.name="Jason"
```

Suppose the Programmer class contains a shared integer property, Total, that gets incremented each time a Programmer instance is created. For example, give the Programmer class a Constructor of:

```
Programmer.Total=Programmer.Total+1
```

The Destructor is:

```
Programmer.Total=Programmer.Total-1
```

Each time a Programmer instance is created or destroyed, the value of the Total shared property is changed to reflect the current count. The value of Total can be accessed from any Programmer instance or from the Programmer class itself.

Adding Constants to Windows

A constant acts like a property but it holds a fixed value for its entire "life." When you create a constant, you give it its value. You can read the constant's value in your code, but you cannot use an assignment statement to change the value of a constant.

You can create constants in REALbasic for windows, modules, and classes. You can also create a local constant inside any method you write. For information about local constants, see the section "Constants" on page 189.

The Scope of Window Constants

A constant for a window has a Scope, just like properties. Scope determines which parts of your application can "see" the constant and read its value.

A constant added to a window can be Public, Protected, or Private in Scope.

- **Public:** The constant can be read by all of the code in your project. To read the value of a Public constant inside the window that owns it, you simply use its name, *constantName*. To read a Public constant from another object's method or event handler outside the window, you use the "dot" notation, i.e., *windowName.constantName.* For example if the window "SaveChangesWindow" has a Public constant called "Accept", you would refer to it as "Accept" from any method or event handler belonging to SaveChangesWindow. However, code belonging to other windows, modules or classes that don't belong to SaveChangesWindow refer to the constant as "SaveChangesWindow.Accept".
- **Protected:** The constant can be read only by code owned by the window and its controls. To read the value of a Protected constant, refer to it by name. For example,

a StaticText control that gets its Text property from a window constant can refer to it by StaticText1.Text=*constantName*. If another window, a class, or a module tries to access a Protected constant, REALbasic will display an informative error message.

- **Private:** A Private constant is like a Protected constant except that any windows subclassed from this window will not be able to access a Private constant. Protected and Public constants are inherited by windows subclassed from the current window. For information about creating subclasses, see the section "Understanding Subclasses" on page 421.

Localizing an Application using Constants

If you need to build separate copies of your application for different platforms and language combinations, you should use constants instead of literal text strings for all your interface items that present text to the user. REALbasic allows you to define different values for a constant for every platform and language combination that you need.

To do this, you use the Localization table in the New Constant declaration area to enter different values for the constant for every platform/language combination that you support. Often these constants are given Global scope—so that they can be referred to by name only—and that is possible only in a module. If you put all of your localization constants in one place it will be easier to maintain them.

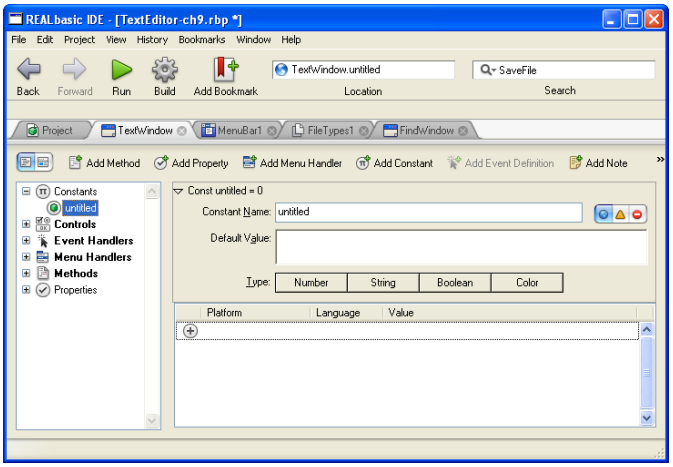
The process is described in detail in the chapter on modules, but you can also use the same process with constants for windows. If you want to use the Localization table for window constants, follow the steps in the section "Adding a Constant to a Module" on page 305.

To add a constant to a window, do this:

1. **Display the Code Editor for the window.**
2. **Click the Add Constant button in the Code Editor toolbar or choose Project ▸ Add ▸ Constant.**

   The Add Constant declaration area appears above the Code Editor area (Figure 218).

Figure 218. The Add Constant declaration area.



3 Enter the name of the constant, its data type, and its value.

Sets its data type by clicking on one of the four Type buttons, Number, String, Boolean, or Color.

If you selected Color, a color patch appears to the right of the Color button with the default color of black. Click it to display the Color Picker to choose the color constant. If you don't see the color patch, expand the IDE window so that there is space to the right of the Color button.

4 Set the Scope of the constant by clicking one of the three Scope buttons.

Your choices for a constant belonging to a window are Public, Protected, and Private, from left to right.

Figure 219. The Scope buttons.

5 (Optional) Use the Localization table at the bottom of the pane to define different values for the constant for different platforms and language combinations.

See the section "Using Constants to Localize your Application" on page 307 for details on the Localization table.

Your application can then access the constant according to the Scope that has been assigned to it. When you want to use the constant as the value of a property in the Properties pane, precede its name with the number sign, #. If you've defined different values for different platforms or languages, the correct value will be used automatically for the version of the application built for that combination of platform and language. You set the default region and language with the Build Settings dialog box. For more information, see the section "Default Region and Language" on page 551.

Adding Methods to Windows

Windows can also have their own methods. The benefit of associating a method with a window is that you can keep code that will be used only with a particular window with that window. For example, suppose you have a window that displays the contents of a document. If the user can save changes to the document in the window, you will need some code that handles saving those changes. Since the window is handling the document, it makes sense that the window should know how to save changes to the document. Therefore, you might want to add a method called *SaveChanges* to the window that handles this. Later, should you decide to use this window for another project, it will have the *SaveChanges* method.

Passing Parameters to Methods

You can pass parameters to methods. Parameters are variables that the method uses internally. When you call the method, you pass values to the method via its parameters. The method can then use these values. Optionally, parameters return new values.

Parameters are declared the same way that properties are (e.g., "Age as Integer"), except that you write out the declaration instead of entering the name and data type in separate fields. To indicate that the method will require parameters when it is called, name and declare the data type of each parameter. If the method requires multiple parameters, the parameter definitions should be separated by commas. For example, if a method requires an integer and a string, you would declare two parameters in the following manner:

```
myInt as Integer, myString as String
```

Parameters are treated as local variables inside the method. This means the method can change the value of a parameter—just as if it was created as a local variable inside the method. In this example, the name "myInt" becomes the name of the local Integer variable inside the method and "myString" becomes the name of the local String variable inside the method.

Passing Arrays

A parameter that you pass to a method can be an array. To pass an array, you follow the name of the array with empty parentheses when you declare the parameter. For example, if you need to pass an array of real numbers, you would declare the array in the following manner:

```
aNums() as Double
```

Since the declaration does not specify the number of elements of the array, the method will accept arrays of any size. You can figure out the number of elements in a particular array by calling the Ubound function inside the method. When you call the method, you simply pass the name of the array you want to pass, without any parentheses.

You can pass multi-dimensional arrays without specifying the number of elements in each dimension, but you need to indicate the number of dimensions. Do this by placing one fewer commas in the parentheses than dimensions. For example, if aNames were a two-dimensional String array, you would declare the array in the following manner:

```
aNames(,) as String
```

Finally, there is one more way of passing a series of values to a method. You can use the ParamArray keyword in the parameter declaration. The ParamArray keyword signifies that any number of values of the specified data type will be passed as parameters to the method. For example, if the method will accept a list of integers, you would declare one Integer parameter using the ParamArray keyword:

```
ParamArray nums as Integer
```

When you call the method, you pass a list of integers, with the integers separated by commas (just as if they had been declared as separate parameters). For example, you can write:

```
myMethod(5,7,2,10)
```

Inside the method, "nums" is an array rather than an integer variable. You can process the values as an array.

Returning Values from Methods

A method can also return a value. A method that returns a value is called a *function*. You create a function using the same process as you use for defining a method; the only difference is that you declare the data type of the value being returned (see Figure 220 on page 266).

The value that a function returns can be an array or just a single value. If your want to return a single value, you specify the data type of the value. If you want to return an array, write empty parentheses after the data type. For example, if you want to return an array of integers, write Integer () as the Return Type. If you need to return a multi-dimensional array, place one fewer commas in the parentheses than dimensions. For example, to return a two-dimensional array of Doubles, write Double(,) as the Return Type.

When you declare a Return Type, your method needs to use the keyword "Return" to indicate the value to be returned. For example, if the method takes an array of numbers, adds them up, and returns the sum into a local variable called "Total", you would need to include the line:

```
Return Total
```

in the function. For more information, see the section, "An Example Method" on page 269.

The Scope of Methods

When you create a method, you indicate how much of your application can "see" the method. You can choose to make the method available only to code belonging to the window that "owns" the method or make it available globally, so that any method or event handler can call the window's method. This is called the *scope* of the method. For windows, you have the following choices:

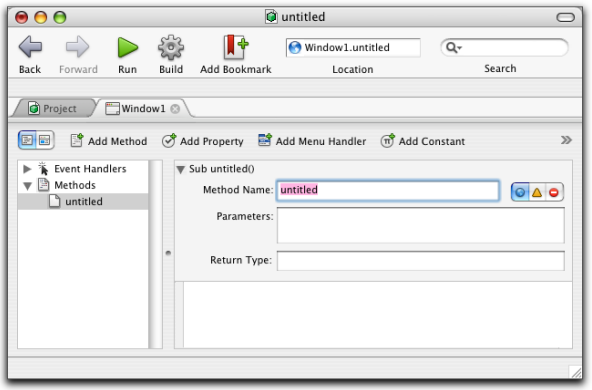
- **Public:** The method can be called from any event handler or method in the application. For example, you would use this if you want a menu handler that belongs to App class to be able to call the method. To call a Public method outside the window that "owns" it, use the syntax *windowName.methodName*. In the window's own event handlers, methods, and controls, you can call it by referring to its name only, *methodName*.
- **Protected:** The method can only be called by the window's event handlers and methods, its controls, and windows subclassed from this window. You would choose Protected if the method performs operations on information in this window and you do not want these operations to be started from outside the window itself. Call a protected method by its name only, *methodName*. If you try to call a Protected method from outside its window, REALbasic will display an informative error message.
- **Private:** The method can be called only by the window's event handlers and methods and controls in the window. It cannot be called by other windows that are subclassed from this window. Windows that are subclassed from this window automatically inherit all its Public and Protected methods. Call a private method by its name only, *methodName*. If you try to call a Private method from outside its window, REALbasic will display an informative error message.

To add a method to a window, do this:

1. **Open the Code Editor for the window.**
2. **Click the Add Method button or choose Project ▸ Add ▸ Method.**

   The Method declaration area appears above the Code Editor area.

Figure 220. The Method Declaration area.



3 Enter the name of the method, its parameters, and, if it will return a value, its return type.

When naming the method, be sure not to use a reserved word. A reserved word is a term that REALbasic uses elsewhere (see "Reserved Words" on page 191). Use the Parameters and Return Type areas to declare the data type of each parameter and the value to be returned (if the method will return a value).

For example, if you were going to pass the value of the TextChanged property declared in Figure 215 on page 257, you would write, "TextChanged as Boolean" in the Parameters area. If you want to pass several parameters, separate each parameter declaration by a comma.

The Return Type is the data type of the value to be returned if your method will be returning a value. The pop-up menu to the right of the Return Type field has a list of common data types, but any valid data type can be defined in the Return Type field.

There are several advanced options available in the parameter declarations area. For more information, see the sections "Passing a Parameter by Value or Reference" on page 270, "Setting Default Values for a Parameter" on page 271, "Setter Methods" on page 273, "Accessing Items of Other Windows" on page 275, and "Constructors and Destructors" on page 275.

4 Select the Scope of the method by clicking a Scope icon.

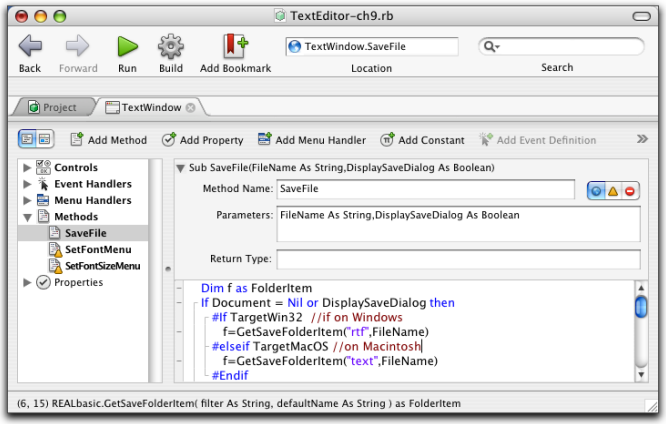
Your choices for a method belonging to a window, from left to right, are Public, Protected, and Private.

Figure 221. The Scope icons.

See the previous section, "The Scope of Methods" on page 266 for information on Scope. A completed method declaration looks like Figure 222.

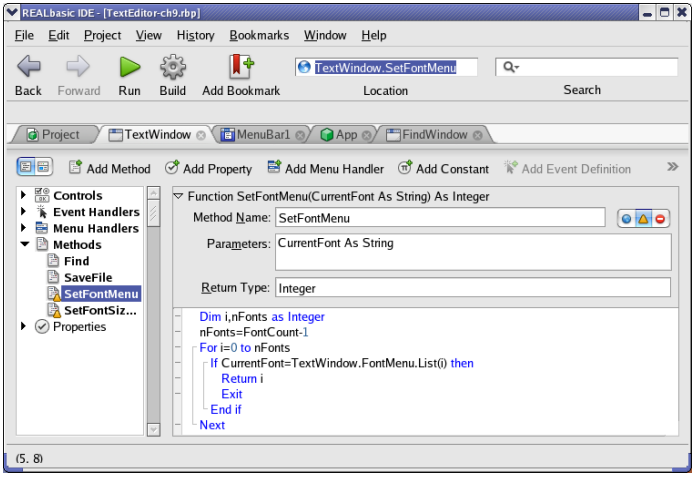
The REALbasic *Tutorial* contains several examples of user-written methods.

Figure 222. The SaveFile method declaration.



If you have declared the method as Private or Protected, the Sub or Function statement indicates the scope of the method, as shown in Figure 223. If it is a Public method or function, then the Sub or Function statement just says "Sub" or "Function."

Figure 223. A Protected method in the Code Editor.



To edit the name, parameters, or return type of a method you have added to a window, do this:

1 **Open the Code Editor for the window that contains the method.**

2 **In the Browser, expand the Methods category to display the list of methods for the window.**

3 **Click on the method to display it in the Code Editor pane.**

4 **If necessary, expand the Method declaration area above the code by clicking its plus sign (Windows) or disclosure triangle (Macintosh or Linux) to the left of the method declaration.**

5 **Edit the name, parameters, return type, or scope as needed.**

To delete a method you've added to a window, do this:

1 **Open the Code Editor for the window that contains the method.**

2 **In the Browser, expand the Methods category to display the list of methods for the window.**

3 **Click on the method you want to delete to select it.**

4 **Choose Edit ▶ Delete or right+click (Control-click on Macintosh) and choose Delete... from the contextual menu.**

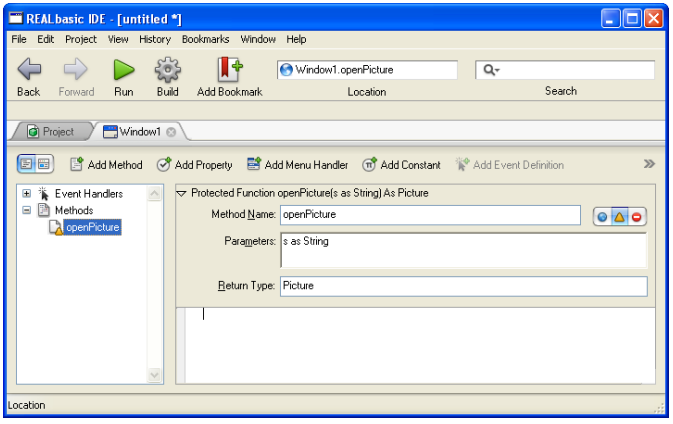
An Example Method

The following example shows how to write a method and call it from any event handler in the window.

Suppose your application needs to allow the user to open pictures and assign the pictures to various objects. You can use a simple method to open the file and return the picture. The event that calls the method can then assign the object to an interface object.

The method will take the name of the file to open as its parameter and return the picture object. Therefore, you declare it as a function, as shown in Figure 224.

Figure 224. The openPicture declaration.



The method consists of the following code (the function declaration is editable in the Method declaration area):

```
Function openPicture (s as String) as Picture
  Dim f as FolderItem
  Dim p as picture
  f=GetFolderItem(s)
  if f.exists then
    p=f.openAsPicture
  else
    MsgBox "Invalid file!"
  end if
  Return p
```

With this method, you can then open a picture file by simply passing it the path to the file as a string. For example, the following line of code displays an image in an ImageWell control by assigning the picture to the Image property of the ImageWell:

```
ImageWell1.image=openPicture("BackgroundImage")
```

Passing a Parameter by Value or Reference

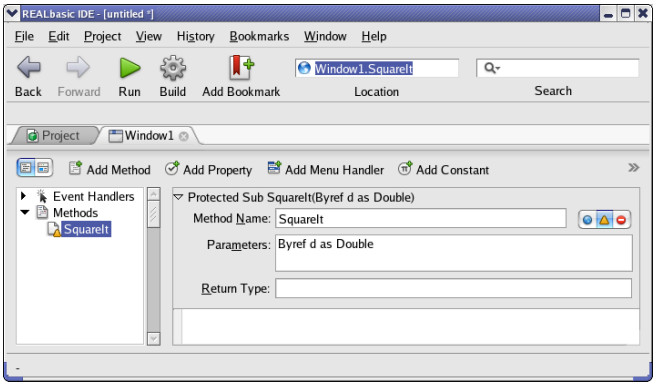
When you define the method's parameters, you can choose between two types of parameter passing: passing by value and passing by reference.

The default is passing parameters by value. With this option, the method receives the value of the parameter and can perform operations on it. The parameter is treated as a local variable within the method. You don't have to do anything special to pass a parameter by value. Passing by value is illustrated in the previous example: a string parameter that contains a pathname is passed to the method and the method uses the passed parameter as a parameter in another function call.

When you use passing by reference, you instead pass a reference to the parameter (i.e., a pointer to the parameter's value). When the method receives a parameter passed by reference, it is free to change the value of the parameter and return the changed value to you after completing the call to the method.

For example, the following method declaration uses one parameter that is passed by reference.

Figure 225. Passing a parameter by reference.



The ByRef keyword in the parameter declaration indicates that the parameter is being passed by reference rather than by value. The method itself consists only of the line:

```
d=d*d
```

That is, the method changes the value of the parameter being passed. Such a method can be called like this:

```
Sub Action ()
  Dim Label as String
  Dim i as Double
  If EditField1.text <> "" then
    i=Val(EditField1.text)
    Label="The square of "+EditField1.text+" is "
    SquareIt(i) //passed by reference
    StaticText1.Text=Label+Str(i)+"."
  Else //no value entered
    Beep
    MsgBox "Please enter a number into the text box."
  End if
```

When you run this method, you will see that the value of i changes after the call to SquareIt. This is not possible when you pass by value.

You can also pass arrays by reference using the same syntax. Simply precede the name of the array with the ByRef keyword. When you pass an array ByRef, the method can change all or some elements of the array.

Setting Default Values for a Parameter

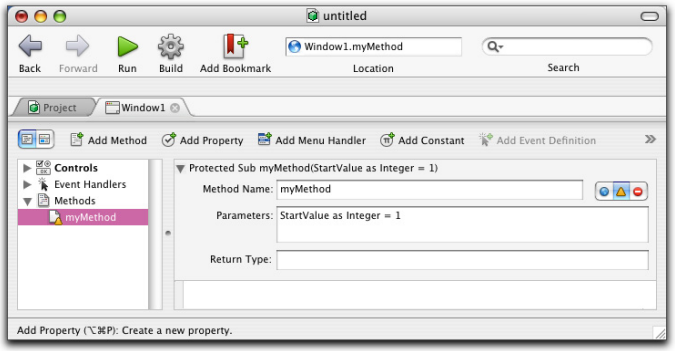
When you declare the parameters to a method, you can optionally provide a default value for each parameter. You do so by making the declaration an assignment

statement in the Method declaration area. For example, if you want to declare the parameter "StartValue" as an Integer and give it value of 1, you would write

```
StartValue as Integer = 1
```

in the Parameters area of the Method declaration area. This is shown in Figure 226.

Figure 226. Declaring a Integer parameter.



When you call the method, any parameter that is provided with a default value in the declaration statement becomes optional.

When you don't provide a default value for the parameter, you must pass an integer value to myMethod whenever you call it. When a default value is provided, you can omit the parameter in the call. When you want to use the default value, simply omit the parameter in the call:

```
myMethod
```

In this case, the default value of 1 will be used.

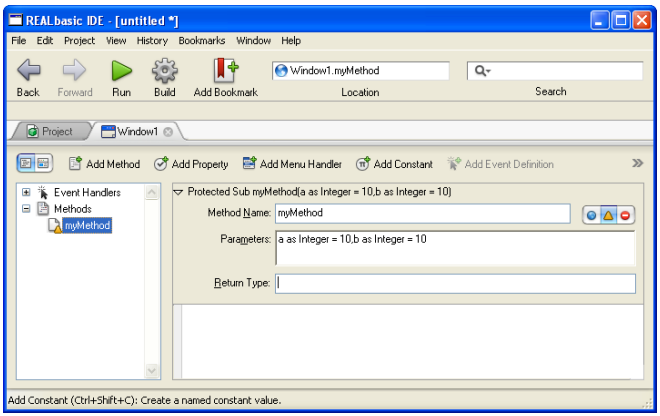
If you want to use another value, pass it as a parameter in the normal way. For example,

```
myMethod(10)
```

In this case, myMethod will use the passed value,10, and ignore the default.

You can provide default values for more than one parameter. For example, the following declaration is valid.

Figure 227. Assigning default values to two parameters.



In this case, you can call the method with the statement:

```
myMethod
```

The two default values will be used. If you want to override the default value for the first parameter, pass one value. For example:

```
myMethod(100)
```

passes the value of 100 to the parameter a. If you want to override only the value of the second parameter, you must pass two parameters. Just pass the default value for the first parameter.

Making a Parameter Optional

If you need to specify that one of the parameters in the method is optional without giving it a default value, use the "Optional" keyword. This modifier precedes the parameter name. An optional parameter does not take on a default value in the called method. If the caller omits this parameter, it will receive the standard default value for its data type.

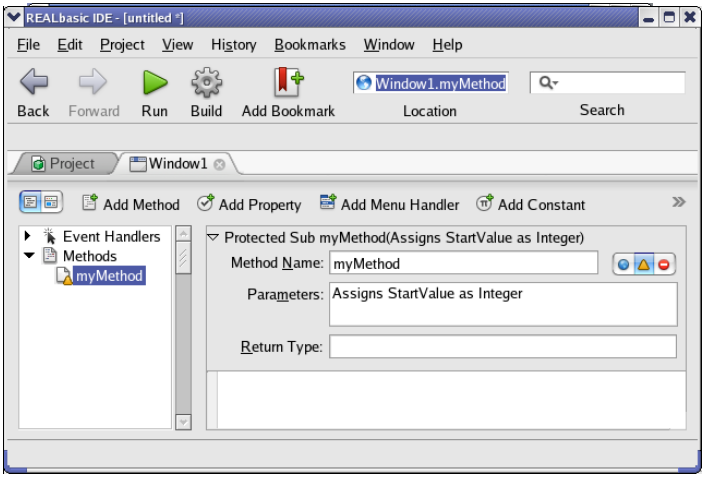
Setter Methods

When you pass a value to a method, you have the option of using the assignment operator. For example, you can pass the value of 10 to a method using the statement:

```
myMethod=10
```

However, the assignment operator can be used only if you use the "Assigns" keyword when you declare the method. In Figure 228, the "Assigns" keyword is used. This syntax must be used whenever you want to use the assignment operator when you pass a value.

Figure 228. Using the "Assigns" keyword in a Method Declaration.

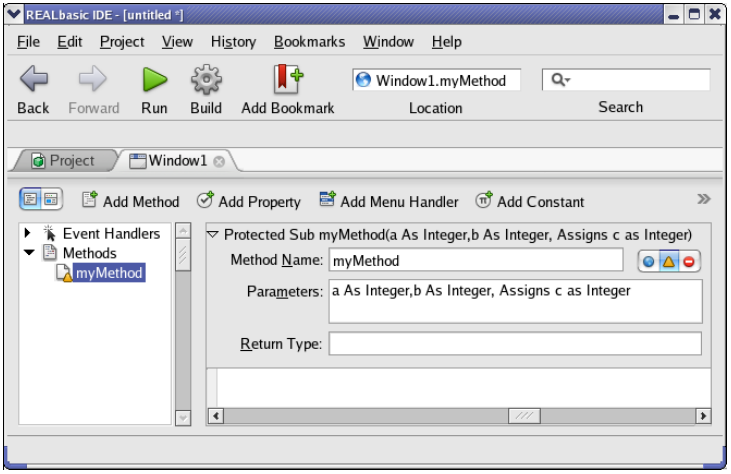


If myMethod is declared as shown in Figure 228, then you can pass a value in the following way:

```
myMethod = 10
```

You can use the "Assigns" keyword with methods that take more than one parameter. When you do so, you must use the Assigns keyword with the last parameter. You can use "Assigns" for only one parameter per method. For example, the following method declaration is valid:

Figure 229. Using "Assigns" with more than one parameter.



To call this method, use the following syntax:

```
myMethod(5,4)=10
```

When you use the Assigns keyword as shown in Figure 229, you cannot use the 'normal' syntax shown below:

```
myMethod(5,4,10) //doesn't work
```

Constructors and Destructors

A *constructor* is a method that runs automatically when the object that owns it is first created. You usually use a constructor to do some type of initialization of the object. A *destructor* is a method that runs when the object is destroyed or goes out of scope.

For information on constructors and destructors, see the section "Constructors and Destructors" on page 443.

Accessing Items of Other Windows

Items in other windows can be accessed using the window name followed by a dot and the control, method, or property name, i.e., *windowName.itemName*.

Of course, this is true only if the Scope of the item you want to access is Public. If the item's Scope has been declared Protected or Private, you cannot access it from another window.

For example, a button in Window1, when clicked, passes the value "Hello" to the "Find" method of Window2. The syntax is:

```
Window2.Find "Hello"
```

The properties of other windows can also be accessed using this syntax. For example, if a button in Window1 should, when clicked, change the title of Window2 to "Hello World", the syntax is:

```
Window2.Title="Hello World"
```

In the case of controls, the control name can then be followed by one of its property names. For example, suppose a button in Window1 will, when clicked, place the text "Hello World" in the Text property of a control called StaticText1 in another window, Window2. The syntax is:

```
Window2.StaticText1.Text="Hello World"
```

That is, the syntax is *windowname.controlname.propertyname.*

The syntax in the previous examples works provided there is only one instance of the target window open. If there are two instances of Window2 open, the code in the previous examples would affect only the first instance of Window2 that was opened.

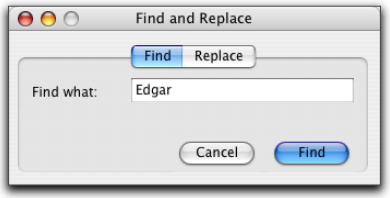
If there can be more than one instance of the target window open, you need to store a reference to that window somewhere so your code will know which instance of the window you are referring to. Where you store this reference depends on how your application works. Suppose you have many instances of a window named

"DocWindow" open that displays the contents of a text document. A button in this window opens a Find window that lets the user enter a value he wishes to search for in that instance of DocWindow. Since there can be many DocWindows open, you will need to store a reference to the specific instance of the DocWindow that opens the Find window in a property of the Find window. You do this by adding a property (let's call it "Target") to the Find window of type DocWindow. When the Find button in an instance of the DocWindow opens the Find window, it can store a reference to the DocWindow in that property. Assuming your application only allows one Find window to be open at a time (perhaps by making the Find window modal), the syntax looks like this:

```
FindWindow.target=Self
```

The Self function returns a reference to the instance of a window (or class) that calls the Self function. In this case, the target property of the FindWindow is being set to a reference to the specific instance of the DocWindow that executed this code. Later, when the user clicks the Find button in the FindWindow, the FindWindow can use the Target property to reference the instance of the DocWindow that opened the FindWindow in the first place.

Figure 230. An Example Find window.



In Figure 230 the FindWindow has an EditField named "FindValue" where the user types what he wishes to find. Let's also assume that the DocWindow has a method called "Find" that, when passed a value, locates that value (if it exists) in an EditField in the DocWindow and highlights the value found. When the user clicks the Find button in the FindWindow, the Find button's Action event handler calls the Find method of the instance of the DocWindow that opened the FindWindow. It does this using the FindWindow's target property and the following syntax:

```
Target.Find FindValue.Text
```

The Target property contains a reference to the DocWindow, so its Find method can be called. In this example, the Find method is being passed the value of the Text property of the FindValue EditField.

The Target property can also be used to change properties of controls in the target window. For example, if you want to disable the Find button in the DocWindow from the FindWindow, you can do so using the following syntax:

```
Target.FindButton.Enabled=False
```

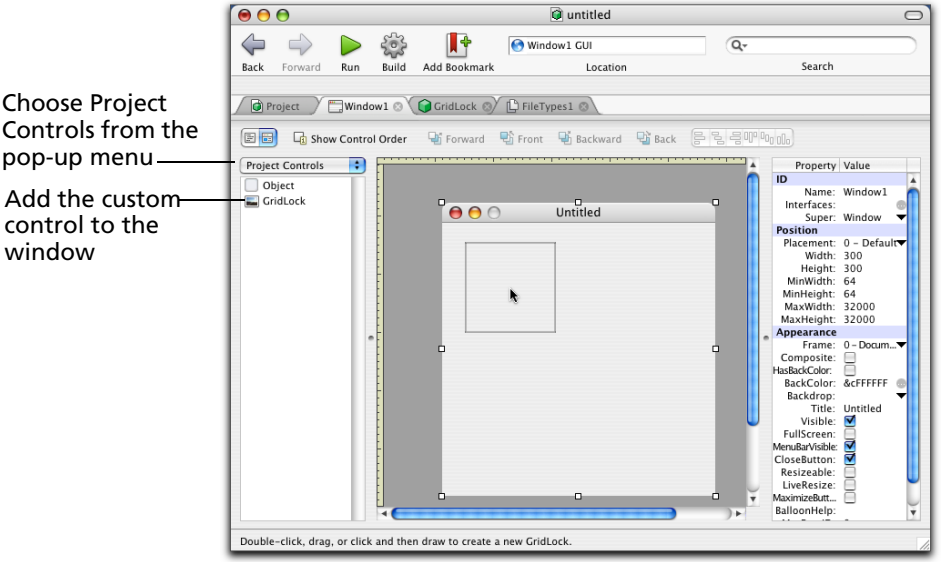
In this example, the Target property of the FindWindow is defined as being of type DocWindow. However, if the FindWindow needs to reference more than one window class, you would define the Target property as type Window to be more generic. This allows the Target property to store a reference to an instance of any kind of window rather than just an instance of DocWindow. However, it also makes the code less readable because it is not clear which windows the FindWindow meant to work with. For this reason, use the generic Window type only when necessary.

Controls

Controls that appear inside a window can have their own code to respond to events directed to them. Unlike windows, you cannot add methods or properties to the controls add to a window from the Controls pane. However, you can create controls that have custom properties, methods, event definitions, and even menu handlers by creating new classes based on the controls.

To do this, you first create a new class that is based on one of the controls and add it to the Project Editor. You declare its Super class to be a control and then add custom methods, properties, or event definitions to the new class. To add an instance of the custom class to a window, use the Controls pop-up menu in the Window Editor to switch to the Project controls and then add the custom control to the window in the usual way.

Figure 231. Adding a custom control based on the Canvas class to a window.



This action adds an instance of the custom control class to a window. See the section, "Understanding Subclasses" on page 421 for information on custom classes based on controls.

Events

Controls, like windows, receive events and have event handlers to respond to the events they receive. For every event a control receives that you can respond to, there is a corresponding event handler. The *Language Reference* contains the complete list of events that windows and controls can respond to. When reading the *Language Reference*, keep in mind the fact that each control inherits properties and events from its parent. For example, most controls are subclassed from the RectControl class, and therefore have all the RectControl class's events and properties.

Creating New Instances of Controls On The Fly

There may be situations where you can't build the entire interface ahead of time and need to create some or all of the interface elements on the fly. This can be done in REALbasic provided that the window already contains a control of the type you wish to create. The existing control is used as a template. For example, if you wish to create a PushButton via code, there must already be a PushButton in the window that you can "clone." Remember that controls can be made invisible, so there is no need for your template control to appear in the window. Once you have created a new instance of the control, you can then change any of its properties.

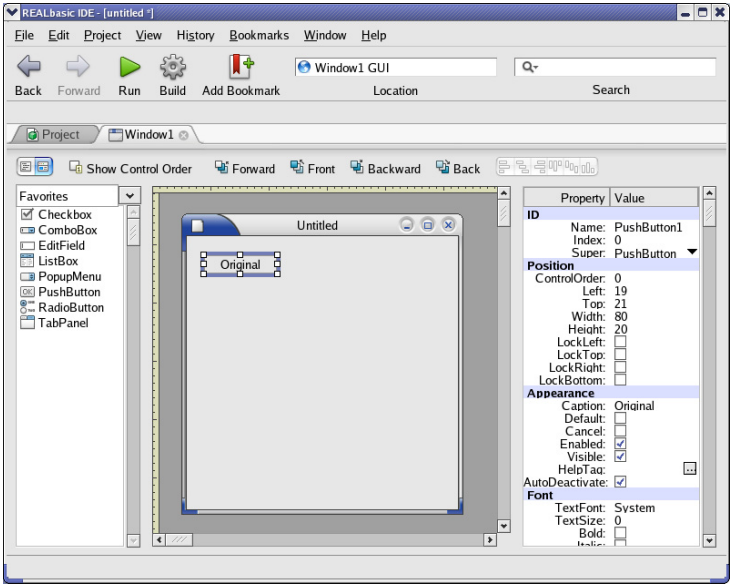
Suppose you need to clone a PushButton that is already in the window, named PushButton1.

To create a new PushButton control on the fly via code, do this:

1 **In the Properties pane for the PushButton, set its Index property to zero.**

When the new controls are created while the application is running, they will, be elements of a control array.

Figure 232. The template control and its properties.



In this example, a new PushButton will be created when the user clicks on the original PushButton.

2 **In the Action event of PushButton1, dimension a variable of type PushButton.**

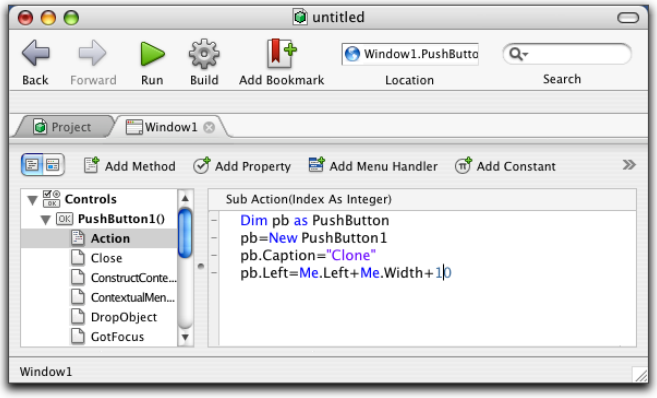
3 **Assign the variable a reference to a new control using the New operator and pass it the name of the template control.**

This example shows a new PushButton being created using an existing PushButton named PushButton1 as a template. When the new control is created, it is moved to the right of the template control:

```
Dim pb as PushButton
pb= New PushButton1  //clone of PushButton1
pb.Caption="Clone"  //change caption just to be clear about this
pb.Left=me.Left+me.Width+10  //move it to the right
```

The Code Editor for PushButton1 should look like this:

Figure 233. Action event that clones PushButton1.



4 Click the Run button.

5 When the application launches, click the "Original" button.

REALbasic creates a new PushButton to the right of Original. You can see it's the variable "pb" from its Caption property. Your screen should show the sequence shown in Figure 234.

Figure 234. The test application before and after creating the cloned PushButton.

If you click "Clone," you will create another clone to the right of the first two, and so on.

Since any new control you create shares the same code as the template control, you may need to be able to differentiate between them in your code. You use the index property of the control to identify which control was clicked. As you can see from Figure 233 on page 280, the Action event is passed the value of the Index property as a parameter. For more information on using the Index parameter, see the following section, "Sharing Code Among An Array of Controls" on page 281.

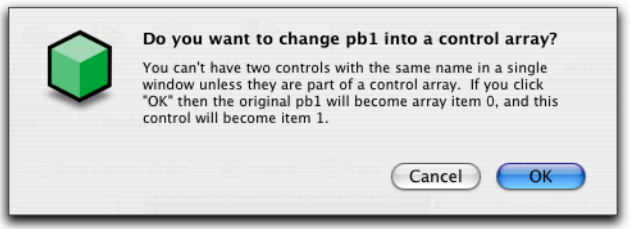
If your code needs to create different kinds of controls and store the reference to the new control in one variable, you can dimension the variable as being of the type of object that all the possible controls you might be creating have in common. For example, if a variable can contain a reference to a new RadioButton or a new CheckBox, the variable can be dimensioned as a RectControl because both RadioButtons and CheckBoxes are derived from the RectControl class. Keep in mind, however, since the variable is a RectControl, the properties specific to a RadioButton or CheckBox will not be accessible. If you need to see which classes of control are common to different controls, see the sections on each control in the *Language Reference*.

Sharing Code Among An Array of Controls

When you have several controls of the same type that all have essentially the same code, the best solution is a control array. A control array allows two or more controls to share the same code. You create a control array by assigning all of the controls the same name and using the Index property to identify the elements of the array of controls.

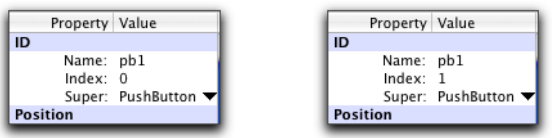
The first time you give a control the same name as another control (that's not already part of a control array), REALbasic will ask you if you wish to create a control array. For example, if you create a PushButton called pb1 and then add another PushButton to the window that you also name pb1, REALbasic will present the dialog box shown in Figure 235.

Figure 235. Creating an array of controls with a dialog box.



If you click OK, REALbasic will assign the first control's Index property the value 0. The control you are renaming will then have its Index property set to 1.

Figure 236. The ID properties of the first two controls in a control array.



Notice that the names of the two controls described in Figure 236 have the same name and are distinguished only by the value of their Index property. After that, any controls in the same window with *the same name* will be assigned the next Index number in the sequence automatically.

The other way to create an array of controls is to enter zero as the value of the Index property of the first control and then assign the first control's name to the second control. REALbasic will then automatically assign 1 to the Index property of the second control, and so on. This process bypasses the dialog box shown in Figure 235 but achieves the same effect.

In the Code Editor, rather than seeing several controls with the same name, the control will appear only once followed by parentheses to let you know it's a control array. All of the controls in the control array share one set of events. Each event in a

control array is automatically passed an Index parameter which tells you which control in the control array actually receives the event.

In the following example, the three RadioButton controls actually consist of a control array. All three controls are named "rb1" and are differentiated by their Index property. In the Code Editor, the value of the Index property is automatically passed to each method. The Action event handler, for example is where you determine which RadioButton was clicked.

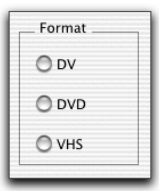
In Figure 237, notice that the Index parameter is automatically added to the method declaration line. Instead of just saying "Sub Action", it says "Sub Action (Index as Integer)". The parameter Index is the index value shown in the Properties pane (shown in Figure 236).

Here is a simple Select Case statement that tests the value of the Index parameter and determines which RadioButton was clicked. This Select Case statement looks at a control array with three RadioButtons.

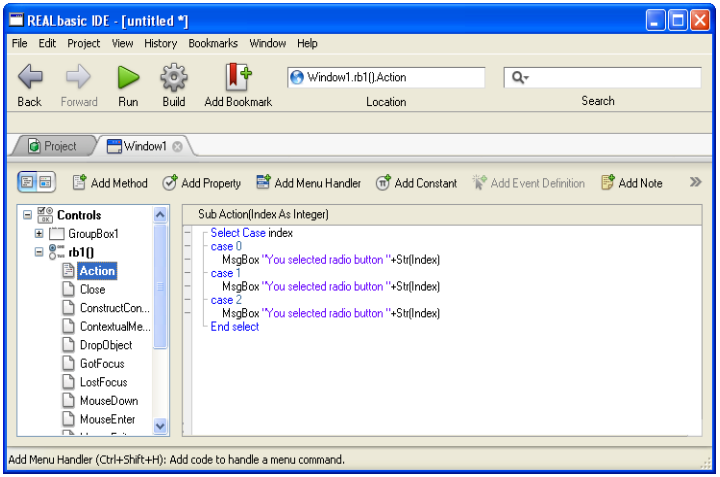
```
Sub Action(Index as Integer)
  Select Case Index
    Case 0
      MsgBox "you selected radio button "+Str(index)
    Case 1
      MsgBox "you selected radio button "+Str(index)
    Case 2
      MsgBox "you selected radio button "+Str(index)
  End select
```

Within each Case, simply insert the code for handling that selection.

Figure 237. A control array and its Action event handler.



When the user clicks a radio button, it sets the value of the Index parameter.



Drag and Drop

Drag and drop is a very important part of the interface of many applications. It extends the concept of the mouse's being an extension of the user's hand. Fortunately, drag and drop is easy to implement in REALbasic. Dragging and dropping of text, pictures, documents, and specified data types is supported.

When something is dragged, a *DragItem* object is created. DragItems have a Text property that is used to hold text being dragged, a Picture property for holding images being dragged, a RawData or PrivateRawData property for holding a specified data type, and a FolderItem property that can contain an object that references a document, folder, or application being dragged. In some cases, you need to populate these properties with data you wish dragged, while in others, the appropriate property will be populated automatically.

A DragItem object can actually contain more than one item and the items don't necessarily have to be of the same type. When you allow the user to drag multiple items, you need to create additional items within the DragItem object and, when the DragItem is dropped, cycle through all items in the DragItem.

DragItems that are dragged to the Desktop or to other applications will act just as you would expect them to. For example, dragging text to the Desktop creates a text clipping file. A DragItem containing a picture that is dragged to the Desktop creates a picture clipping file.

Dragging Text From EditFields

Only text in EditFields, rows in ListBoxes, portions of Canvas controls, images in ImageWells, and Windows can be dragged. If you have never implemented drag and drop before, this may sound like a limitation, but in fact, it isn't. These controls are the only types of objects that can be dragged in other applications that support drag and drop.

The text in an EditField can be dragged automatically without any coding necessary, provided that the MultiLine property of the EditField is True. A DragItem object is automatically created and the text the user is dragging is placed in the Text property of the DragItem.

Dragging A Row From a ListBox

In order for the user to be able to drag a row from a ListBox, the EnableDrag property of the ListBox must be set to True. When the user attempts to drag a row, the DragRow event handler of the ListBox executes and is passed the DragItem that was created and the row number of the row being dragged. You then have to populate the Text property of the DragItem passed. Finally, since the DragRow event handler is actually a function, your code must return True to allow the drag to occur. Returning False or returning nothing at all prevents the drag. This example code

from the DragRow event handler of a ListBox handles dragging a row from the ListBox:

```
Function DragRow(Drag as DragItem, Row as Integer)
 Drag.Text=Me.List(Row)+chr(13) //get the text
 Return True //allow the drag
```

Dragging from an ImageWell

Drag and drop to or from an ImageWell is simple. When dragging from an ImageWell control you must:

- Create a DragItem object using the NewDragItem method,
- Load the data to be dragged into the new DragItem instance,
- Call the DragItem's Drag method to allow the drag to occur

The DragItem object is the container that holds the dragged image. NewDragItem is a method that takes as its parameters the left, top, width, and height of the drag rectangle you want displayed when the user begins the drag. The Drag method is called when the data to be dragged is loaded in the DragItem. You place this code in the ImageWell's MouseDown event handler, since the user presses the mouse button to initiate the drag.

```
Function MouseDown(X as Integer,Y as Integer) As Boolean
 Dim d as DragItem
 d=NewDragItem(Me.left,Me.top,Me.width,Me.height)
 d.picture=Me.image
 d.Drag //Allow the drag
```

Dragging from a Canvas Control

Dragging the backdrop image from a Control is the same as dragging the image from an ImageWell. You place this code in the Canvas control's MouseDown event handler, since the user presses the mouse button to initiate the drag.

```
Function MouseDown(X as Integer,Y as Integer) As Boolean
 Dim d as DragItem
 d=NewDragItem(Me.left,Me.top,Me.width,Me.height)
 d.Picture=Me.backdrop //populate DragItem with data
 d.Drag //Allow the drag
```

When you program the Drop, you will assign the Picture property of the DragItem to a property of the control receiving the drag.

Dropping

In order for the user to be able to drop a DragItem on a control or window in your application, the control or window must have previously indicated that it will accept the kind of data the user wishes to drop on it. There are four methods that any control can call to indicate the type or types of data that can be dropped on that

control. You can indicate that the control or window can accept more than one data type by using as many methods in Table 11 as appropriate.

Table 11: Methods that control the type of data that can be dropped on a control.

Name	Description
AcceptTextDrop	Indicates that the control or window will accept text being dropped on it.
AcceptPictureDrop	Indicates that the control or window will accept a picture being dropped on it.
AcceptFileDrop (Type)	Indicates that the control or window will accept files (of the type or types passed) being dropped on it. The file types must be defined as file types for this project in the File Type Set or via the FileType class. See the section "Using The File Types Editor" on page 374 for information on defining file types.
AcceptRawDataDrop (*Type*)	Indicates that the control or window will accept the data type specified by the four-character Type code, e.g., AcceptRawDataDrop ("mytp"). Use this to drag and drop data in special formats or to control where text strings can be dropped.

Typically, the control or window will call one or more of these methods in its Open event handler. However, if a control or window only accepts items dropped on it under certain conditions, these methods can be called once those conditions are met even after the window is opened.

In most cases, when something acceptable is dropped on a control or window, the target's DropObject event handler is executed. This event handler is passed a DragItem object that represents the item being dropped. If the target has indicated that only some kinds of data are acceptable, your code can get the data from the appropriate property of the DragItem. The data types are shown in Table 12:

Table 12: DragItem properties that indicate the type of object dropped on a control.

Name	Description
FolderItem	Represents an application, folder, or document that has been dropped.
Picture	The picture, if any, that has been dropped.
Text	The text, if any, that has been dropped.
RawData (Type)	Data of the Type indicated by the four-character type code. Data of the format indicated by the Type code is returned as a string. Supported only on Macintosh and within the REALbasic application on Windows.
PrivateRawData (Type)	The data (of the Type specified) being dragged. Type is a four-character resource type or programmer-defined four-character code. This data cannot be dragged to another application.

If more than one kind of data can be dropped, the code in the DropObject event handler needs to determine what kind of data has been dropped. This can be done using these functions of the DragItem:

Table 13: DragItem functions indicating the type of data that has been dropped.

Name	Description
FolderItemAvailable	Returns True if one or more applications, folders, or documents have been dropped.
PictureAvailable	Returns True if a picture was dropped.
TextAvailable	Returns True if text was dropped.
RawDataAvailable (Type)	Returns True if data of the type indicated by the four-character Type code was dropped.

Dropping Items On EditFields

Text dropped on a multiLine EditField is placed in the EditField at the insertion point automatically. The EditField's DropObject event handler is not called. Pictures and files dropped on a multiLine EditField, however, cause the DropObject event handler to execute. For example, if you want to be able to drop a text file on an EditField and have the contents appear in the EditField, you need to get the FolderItem from the DragItem that is passed to the EditField's DropObject event handler and read the contents of the file.

In this example, an EditField has been set up to accept text files dropped on it. *Me* is the generic representation for the object that owns the event handler:

```
Sub DropObject(Obj as DragItem)
 If Obj.FolderItemAvailable then
  Obj.FolderItem.OpenStyledEditField Me
 End if
```

Since more than one file can be dropped at a time, you need to use the NextItem function of the DragItem to determine if there is another file that has been dropped. The NextItem function also changes the FolderItem property of the DragItem to the next file. The last example, modified to handle more than one file dropped on it, looks like this:

```
Sub DropObject(Obj as DragItem)
 If Obj.FolderItemAvailable then
  Do
   Obj.FolderItem.OpenStyledEditField Me
  Loop Until Not Obj.NextItem
 End If
```

Dropping Items on ListBoxes

If you want to drag text to a ListBox, you need to tell the ListBox to receive dragged items by placing the following line in its Open event handler (or another event handler that runs prior to the user's drag and drop):

```
Me.AcceptTextDrop
```

Next, you need to tell the ListBox what to do when dragged text is coming its way. You do that in its DropObject event handler:

The following DropObject event handler determines whether the dragged item has text; if it does, it creates a new row and assigns the dragged item's text property to the new row.

```
Sub DropObject (Obj as DragItem)
 If Obj.TextAvailable then
  Me.AddRow(obj.text)
 end if
```

Dropping Items on ImageWells and Canvas controls

To allow the user to drop a picture or a PICT document that is being dragged from the desktop, add the following statements to another ImageWell or Canvas control's Open event handler:

```
Me.AcceptPictureDrop
Me.AcceptFileDrop("image/x-pict")
```

The second statement assumes that the PICT file type has previously been added in the File Types Set Editor or via the FileType class via the language (see "Using The File Types Editor" on page 374 for more information).

Next, you need to tell the ImageWell or Canvas what to do when the user drops either type of DragItem. You do this in the DropObject event handler. The following works for an ImageWell.

```
Sub DropObject (Obj as DragItem)
 If Obj.PictureAvailable then
  ImageWell1.Image=Obj.Picture
 Elseif Obj.FolderItemAvailable then
  me.image=Obj.FolderItem.OpenAsPicture
 End if
```

It tests whether the DragItem is a picture or a file (FolderItem). If it's a picture, it assigns the Picture property of the DragItem to the Image property of the ImageWell; if it's a FolderItem, it opens the document as a PICT file and assigns that to the ImageWell's Image property.

If you want to assign the dropped object to a Canvas control's BackDrop property, the DropObject event handler is practically identical:

```
If Obj.PictureAvailable then
 Canvas1.Backdrop=Obj.Picture
Elseif Obj.FolderItemAvailable then
 Canvas1.Backdrop=Obj.FolderItem.OpenAsPicture
End if
```

You can also drag a text item to a Canvas control and use the DrawString method in the Graphics class to draw the text. To allow the drag, use the line:

```
me.AcceptTextDrop
```

in an event handler that runs prior to the user's drop. Next, test for text in the DropObject event handler. The following code accepts dragged text and writes it at a specified location:

```
If Obj.TextAvailable then
 Canvas1.Graphics.DrawString(obj.text,20,20,50)
End if
```

You will need to update the data using the Paint event handler if you want the data to persist.

RawData and PrivateRawData Properties

The RawData and PrivateRawData properties allow you to drag and drop other data types. Each property takes a four-character Type that corresponds to the four-character resource code of the format you wish to drag. For example, if you want to support dragging sound clippings, you would use "snd " as the four-character code.

Regardless of format, the data is stored in the DragItem as a string and the DropObject event handler would pass the data to a property that stores a string. The REALbasic control or window that receives the DragItem must be capable of working with the format. In most cases, that means that the control is a custom control, uses toolbox calls, and/or uses a plug-in that manages the data.

For example, to allow the user to drop a 'snd ' resource on a control, you would write

```
acceptRawDataDrop("snd ")
```

in the control's Open event handler. The DropObject event handler would pass the data to a property that stores a string. To actually play the sound, the control would have to make toolbox calls or pass the data to a plug-in since REALbasic doesn't provide a way to pass sound data into a Sound class.

You can also use RawData or PrivateRawData to manage internal drag and drop. If you make up a format type that is different than the name of any resource type, you

can use it to control which objects are dropped on a control. For example, the DragRow event handler of a ListBox:

```
Function DragRow(drag as DragItem, row as Integer) as Boolean
 Drag.RawData("mytp")=me.List(row)
 Return True
```

uses the Type "mytp" to define the format. The DragItem is assigned the row in a ListBox that is being dragged. Although the row is an ordinary string, this DragItem cannot be dropped on a control unless it accepts dragged data in the "mytp" format. In this manner, for example, you can create a control that will only accept a DragItem from a specific control and reject dragged items from the desktop, other applications, and other controls.

The ListBox receiving the data would use the statement:

```
me.AcceptRawDataDrop("mytp")
```

to permit the data to be dropped. The DropObject event handler assigns the data to a new row:

```
If obj.RawDataAvailable("mytp") then
 me.AddRow(obj.RawData("mytp"))
End if
```

The PrivateRawData property works the same way, except that the DragItem cannot be dragged to the desktop or to any other application.

Menus and Menu Items

Menus and menu items are handled in a way similar to controls and are just as object-oriented. This means that you can handle menu selections at the application, window, or even control level. When you create a new Desktop Application project, REALbasic automatically includes one menubar object in the project, named MenuBar1. This is the default global menubar for the entire application. By default, this is the only menubar that is used for the application.

You can also add additional menubars to your project and assign menubars to windows. On Windows and Linux, the menubar "belongs" to the window and is always used when the window is visible. On Macintosh, the window's menubar replaces the current menubar when the window becomes active.

When the user selects a menu item or presses the menu item's command key equivalent, an event occurs much in the same way that an event occurs when the user clicks on a PushButton. In this case, the event handlers are instead called *menu handlers*. For information on creating menus, see "Adding Menus and Menu Items" on page 154 of chapter 3.

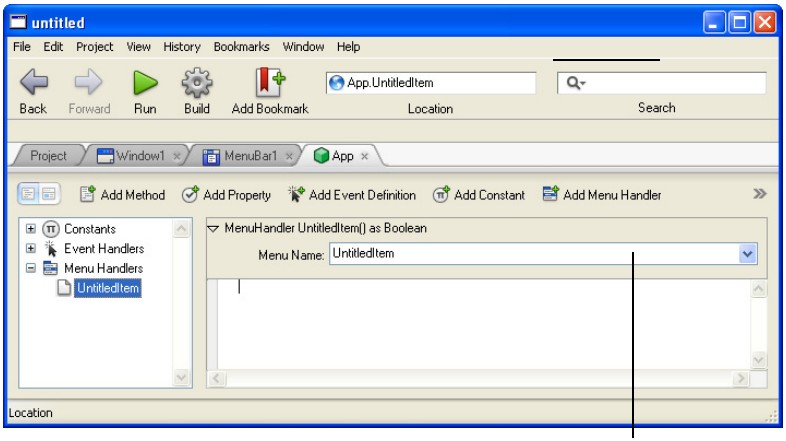
Adding Code To a Menu Item

To add a menu handler to the current window or class, do this:

1. **Open the Code Editor for the window or class.**
2. **Click the Add Menu Handler button in the Code Editor toolbar or choose Project ▸ Add ▸ Menu Handler.**

   The Menu Handler declaration area appears in the Code Editor.

Figure 238. The Menu Handler declaration area.



Menu Item pop-up menu populated with existing menu items

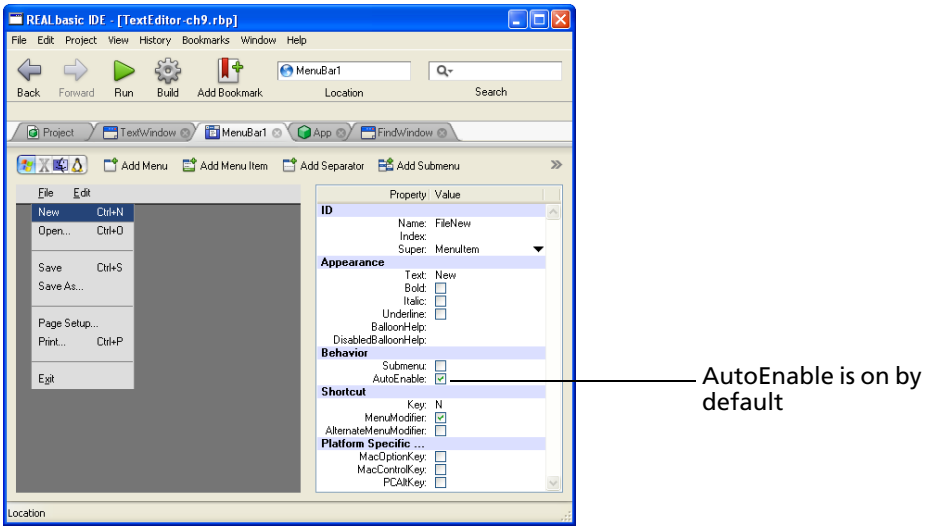
3. **Choose a menu item from the Menu Item pop-up menu.**

   The Code Editor appears, with the new menu handler selected for code entry.
4. **Click in the Code Editor area and enter the code that will execute when the user chooses the menu item.**

Enabling Menu Items

Each menu item object has a boolean AutoEnable property. When the AutoEnable property is set to True, the menu item is enabled unless you explicitly disable it. You set the AutoEnable property in the Behavior group in the menu item's Properties pane.

Figure 239. The AutoEnable property in a menu item's Properties pane.



Unless you turn the AutoEnable property off, a menu item that has a menu handler will be enabled automatically. You should take advantage of the AutoEnable property of menu items that should be enabled all the time, such as a menu command that creates a new document or opens an existing document.

If you turn off AutoEnable, the menu item is disabled by default. You have the responsibility of enabling it whenever it is appropriate for it to be enabled. You do so with the EnableMenuItems event handler. You will want to turn AutoEnable off and use the EnableMenuItems event when a menu item should be enabled only under certain conditions. For example, if you want a "Save" menu item to be enabled only when the user has made changes to a document since the last Save, you would use the EnableMenuItems event to determine when to enable and disable the menu item.

When the user clicks on a menu to select a menu item or presses a keyboard equivalent, an EnableMenuItems event occurs. The purpose of this event is to give you the opportunity to determine whether the menu item being selected should be enabled or disabled based on conditions at the time. REALbasic first checks to see if the control that has the focus is capable of handing menus. If it is, it is sent an EnableMenuItems event. Then, assuming a window is open, the frontmost window is sent the EnableMenuItems event. Finally, the application object is sent the EnableMenuItems event.

Menu items are objects just like controls. Consequently they have an Enabled property that determines if the menu item is enabled or disabled. You can enable a menu item by setting this property to True or by calling the menu item's Enable method from within an EnableMenuItems event.

For example, this EnableMenuItems event handler is checking a property called *Changed* to determine if the Save menu item should be enabled:

```
Sub EnableMenuItems()
 If Me.Changed Then
  FileSave.Enable
 End If
```

Handling Menu Items From Individual Controls

If the control that has the focus is capable of handling menus, its EnableMenuItems event handler will be executed. If the menu item selected is then enabled and the user selects it, the control's menu handler for the selected menu item (if it has one) will be executed. In order for a control to be able to handle menu items, it must be able to receive the focus and it must be based on a class you have added to your project rather than created by dragging a control from the Controls list. See Chapter 9, "Creating Reusable Objects with Classes" on page 419 for more information on handling menu items from control classes.

Handling Menu Items When a Window Is Open

You already know that when the user attempts to select a menu item, the frontmost window's EnableMenuItems event handler is executed followed by the application object's EnableMenuItems event handler. This gives you the opportunity to determine if conditions in the current window are right to permit the user to select various menu items. When the user selects the menu item, REALbasic executes the frontmost window's menu handler for the selected menu item (assuming one exists) followed by the application object's menu handler.

Handling Menu Items When No Windows Are Open

When there are no windows open, the EnableMenuItems event is sent to the Application object. This condition can occur on Macintosh. Assuming the Application object enables the menu item and the user selects the menu item, the Application object's menu handler for the selected menu item (if one exists) is executed.

When you create a new project, a class based on the Application class is included in your Project Editor. This is the App class.

If the user closes all windows in your application and then decides to use the menu item to create a new window, you must enable such a menu item in the Application object, since no windows are available to enable the menu item and handle the menu selection (assuming you are not using the menu item's AutoEnable property). Open the App class's Code Editor from the Project Editor, expand the Events item, and select the EnableMenuItems event. Add code that enables the menu item under the correct circumstances.

Creating New Menu Items On The Fly

This is handled in a way that is similar to how you create controls on the fly. A menu item that can act as a template must already exist. This menu item will effectively be "cloned." You can then change the clone's properties such as the Text, keyboard shortcut, etc. The difference is that the menu items must have an index value in their Index property in order to be used as a template. Assign a zero to the Index

property of the menu item to create a menu item array. The menu handlers for the menu item will then be passed an Index parameter that allows you to determine which menu item was selected. If you don't assign an index value, you will have no way of knowing which menu item was passed. Once you have setup the template menu item, you can create new menu items on the fly using the New operator. This example creates a new menu item based on an existing menu item named "WindowItem."

```
Dim t as MenuItem
t=New WindowItem
```

Remember that once you have created a menu item array, you must refer to the items in that array as array elements. For example, to enable the first menu item (item zero from the WindowItem example), use the following syntax:

```
WindowItem(0).Enabled=True
```

If you wish to be able to programmatically remove menu items you have created dynamically, you need to store the reference that was returned when you created the menu item. You can then use this reference to remove the menu item by calling the Close method. For example, you are storing references to the menu items in a module property array called "WindowRefs." You can then remove a particular dynamically created menu item (the item stored in the fourth array element in this case) using this syntax:

```
Window Refs(4).Close
```

Displaying a Contextual Menu

In addition to the menus that you add to your application via the Menu Editor, you can also add menus that display only as contextual menus. There are several ways to display a contextual menu in REALbasic.

The preferred way is to use the ConstructContextualMenu and ContextualMenuAction events of the Window class (for windows) and for the RectControl class (for all visible controls).

The ConstructContextualMenu event handler "figures out" when the user has requested a contextual menu, regardless of how he did it. It could be a right+click (Windows and Linux), ctrl-click (Macintosh) or the user has pressed the contextual menu button on a Windows keyboard or the user has pressed Shift+F10. It also takes care of cross-platform differences in when the contextual menu is displayed. On Windows and Linux, it is displayed in the MouseUp event but on Macintosh, it is displayed in the MouseDown event.

The ConstructContextualMenu event is passed a "base" menu item, which serves as the "menu bar" for the contextual menu. You add your contextual menu items to the base. It is also passed the X and Y coordinates of the mouse click, so you know where to draw the contextual menu.

You can build the contextual menu in this event. To display the contextual menu, simply return True.

The following is a simple ConstructContextualMenu event handler. It constructs a contextual menu with two items and displays it.

```
Dim m1 as New MenuItem
Dim m2 as New MenuItem

m1.Text="Import"
m2.text="Export"

base.append m1
base.append m2

Return True  //display the contextual menu
```

To handle the menu selection, you can use a Select Case statement in the ContextualMenu Action event. The following Select statement in the ContextualMenuAction event handler inspects the selected menu item, which is passed in as the HitItem as MenuItem parameter.

```
Select Case hititem.text
 case "Import"
  MsgBox "You chose Import"
 case "Export"
  MsgBox "You chose export"
End select

Return True
```

Displaying a Menu as a Contextual Menu

In some applications, you will want to give the user the option of choosing a menu item from one of the application's menus from a contextual menu. For example, you may want to let users choose the Cut, Copy, and Paste menu items from the "regular" Edit menu.

You use the PopupMenu method of the MenuItem class to do exactly this. If you want to display the contextual menu when the user when the user right+clicks an object, use the IsContextualClick function to test whether the user has right+clicked (or Control-clicked on Macintosh) in the correct area. Then call the Popup method.

By default, the contextual menu will appear where the mouse button was depressed. If you want it to appear elsewhere, pass the PopupMenu method the X and Y coordinates of the desired point on the screen. The top-left corner of the monitor is the 0,0 point.

The following example is in the MouseDown event handler of an EditField. If the user right+clicks, the Edit menu is displayed as a contextual menu. The selected

menu item is returned in the MenuItem, m. Before the MenuItem is returned, the selected MenuItem's Action event occurs, so you can handle the menu selection there, just as if the user chose the menu item from the menu in the menu bar or window title bar

```
if IsContextualClick then
  Dim m as New MenuItem
  m=EditMenu.Popup
End if
```

Yet another way of displaying a contextual menu is to build it with the ContextualMenu control and detect a request for it to be displayed with the IsContextualClick function.

Here is an example. The following code in a ContextualMenu control's Open event handler populates the control:

```
Dim s as String
Dim i,last as Integer
s="Undo,-,Cut,Copy,Paste" //dash adds a separator below Undo
last=CountFields(s,",")
For i=1 to last
 Me.addRow NthField(s,",",i)
Next
```

The following code in a control or window's MouseDown event handler displays the contextual menu when the user right+clicks in the control or window (or Control-clicks on Macintosh):

```
If IsContextualClick then
 ContextualMenu1.open
 Return True
End if
```

When the user chooses a menu item, the ContextualMenu's Action event handler runs. The selected menu item is passed to the event handler, so you know which item the user has chosen.

Classes

Classes can be used to create custom controls that can also respond to the user. For more information on using classes to create custom controls, see Chapter 9, "Creating Reusable Objects with Classes" on page 419.

CHAPTER 6

Adding Global Functionality with Modules

Object-oriented programming can be very efficient but you may find occasions when you need to add constants, methods, functions, and properties that are not associated with any one object. For example, you might need to add some custom financial functions that will be called from many different places within your application. You may need to store values that are associated with those functions. In most cases, when you need to add a constant, method, function, or property that isn't associated with any particular object and needs to be accessible globally, a module is the perfect place to add it.

Constants can have a special purpose when added to modules. They can be used to localize the interface elements of an application. This usage is described in the section "Using Constants to Localize your Application" on page 307.

In this chapter, you will learn what modules are, when to use them, and how to add methods and properties to them.

Contents

Understanding Modules

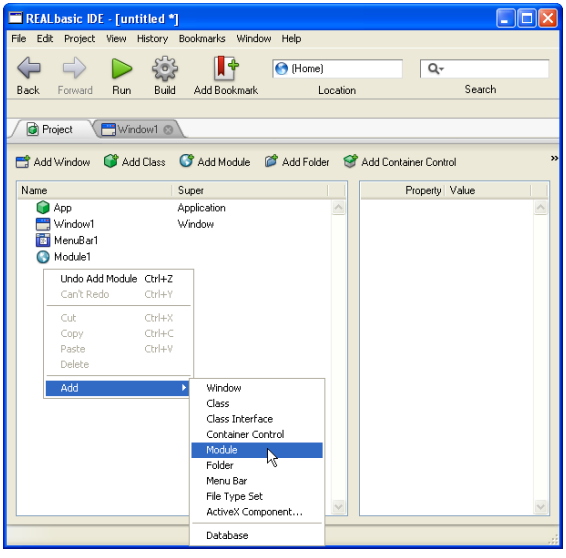
In REALbasic's object-oriented environment, methods, constants, and properties are usually part of another object. Protected methods, constants, and properties associated with objects are only accessible through those objects.

Modules are not objects. A module is not a class and does not have a Super Class. You don't instantiate modules with the New command in order to access them. Once you add a module to your project and then add global or public methods, constants, or properties to it, those objects are immediately accessible outside the module. The exceptions are protected methods, properties, and constants that belong to the module. Protected items are accessible only from other methods in the same module.

Adding A New Module

You can add a new module to your project by clicking the Add Module button in the Project Editor or by choosing Project ▶ Add ▶ Module. You can also use the Project Editor's contextual menu to add an item to the project. Right+click (Control-click on Macintosh) in the Project Editor and choose Add ▶ New Module from the contextual menu.

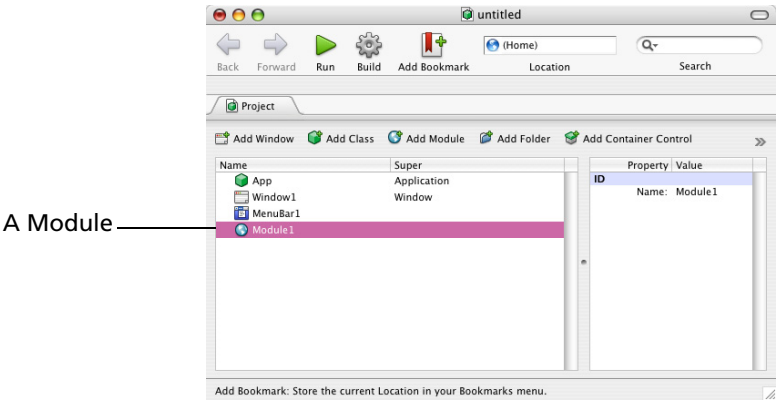
Figure 240. Adding a module to the project with the contextual menu.



The new module appears in the Project Editor with a default name (the first module you add will be named "Module1," for example). You can use the Properties pane to rename the module to something more appropriate. For example, if the module will contain your financial functions, you might name it "Financial."

Modules can only contain methods, constants, and properties. You create and modify them with the Code Editor. To access the Code Editor for a module that is not already open, simply double-click the module's name in the Project Editor. If it is open, click on its tab in the Tab bar. Modules can be identified by their special icon in the Project Editor.

Figure 241. A module in the Project Editor



Scope of a Module's Methods, Properties, and Constants

When you add a method, property, or constant to a module, you need to set its Scope attribute. The Scope of a method, property, or constant determines which other items in the project can access it. There are three possible values:

- **Global:** A Global method, property, or constant is available to code throughout the application. The Global scope is available only for items in modules. For example, you can use a global property to store a piece of information that needs to be available to several different windows and to the application as a whole even if no window is open. When your code needs to access a global method, property, or constant you simply reference it by name from anywhere in the application. If you wish, you can also use the syntax for Public items.
- **Public:** A Public method, property, or constant is also available to code throughout the application. However, when you need to access a public item outside of the module, you use the "dot" notation and precede its name with the module's name. For example, if you declare a Public property, myPublicProperty, in Module1, you call it by referring to "Module1.myPublicProperty" outside Module1. Within a

method or event handler belonging to Module1, it is in scope, so you can access it by referring to "myPublicProperty".

- **Protected:** A Protected method, property, or constant is available only to other code within the module. It is "invisible" to the rest of the application. When code inside the module needs to access a Protected method, property, or constant, you simply reference it by name. If you try to access a Protected method, property, or constant outside of the module, REALbasic will display an informative error message indicating that the item is out of scope.

  In windows and classes, the most restrictive scope is Private. Private scope is the same as Protected except that any windows or classes that are subclassed from the current window or class do not inherit the method, property, or constant being declared. Since a module is not a class and cannot be subclassed, there is no reason to make this distinction.

Adding Methods to Modules

Adding methods to modules is done in the same way you add methods to a window. For more information, see the discussion about creating methods in the section, "Adding Methods to Windows" on page 264.

To add a method to a module, do this:

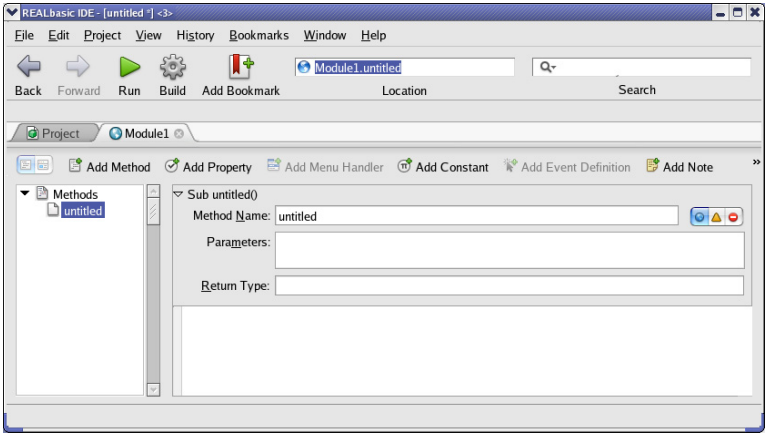
1. **If the Code Editor for the module is not already open, double-click the module's name in the Project Editor.**

   The module's Code Editor appears.

2. **Click the Add Method button in the Code Editor toolbar or choose Project ▶ Add ▶ Method.**

   The Method Declaration area appears above the Code Editor area (Figure 242).

Figure 242. The Method declaration area.



3. **Enter the method name and parameters.**

When you enter the parameters, indicate the data type of each parameter. For example, if you are going to pass the value of a real number that you'll refer to as "X" in the method, write "X as Double" in the Parameters area. If you want to pass several parameters, separate each parameter declaration by a comma. If you want to pass an array, write empty parentheses after the name of the array.

4 If the method is going to be a function, choose the data type of the value the function will return in the Return Type field.

The value that a function returns can be an array or just a single value. If you want to return an array, write empty parentheses after the name of the data type in the Return Type area. For example, if you want to return an array of integers, write "Integer ()" as the Return Type.

There are several advanced options available in the parameter declarations area. For more information, see the sections "Passing a Parameter by Value or Reference" on page 270, "Setting Default Values for a Parameter" on page 271, "Setter Methods" on page 273, "Accessing Items of Other Windows" on page 275, and "Constructors and Destructors" on page 275.

5 Choose the Scope of the method by clicking one of the three Scope buttons.

Your choices, from left to right, are Global, Public, and Protected. Note that elsewhere in REALbasic, the icons stand for Public, Protected, and Private.

Figure 243. The Scope buttons.

For more information, see the previous section "Scope of a Module's Methods, Properties, and Constants" on page 299."

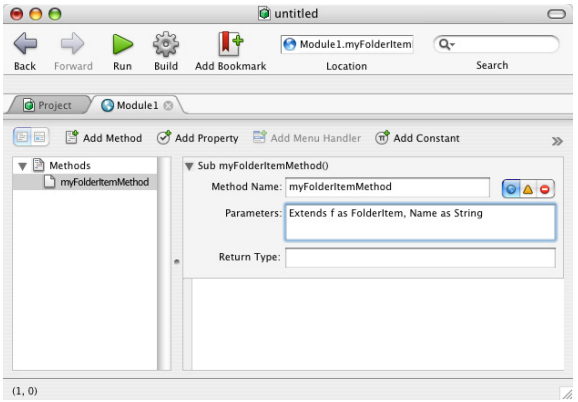
Class Extension Methods

A "class extension method" is a method that can be called using syntax that indicates that it belongs to another object. For example, you can add a method that is called from any FolderItem object that saves in a particular format. After you add the class extension method to a module, you can call it as if it were built into the FolderItem class.

To define a method as a class extension method, use the "Extends" keyword prior to the first parameter. The data type of the first parameter is the object type from which the method must be called. In other words, the use of the "Extends" keyword indicates that the parameter is to be used on the left side of the dot (".") operator in a calling statement.

For example the following example creates a method that will be called as a method of the FolderItem class.

Figure 244. Declaring a class extension method of the FolderItem class.



The Extends keyword can be used only for methods that reside in modules. As long as the module is in the project, the class extension method can be called from anywhere in the project. The Scope of this method is Global.

To use the class extension method in another project, simply import the module into that project. For information about importing modules, see the section “Importing and Exporting Modules” on page 317.

For an example of a class extension method, see the section, “Extending Classes” on page 442.

Adding Properties to Modules

Adding properties to modules is done in the same way you add properties to a window. You set the Scope of a property to determine whether the property will be available to the whole application or only to code within the module.

To add a property to a module, do this:

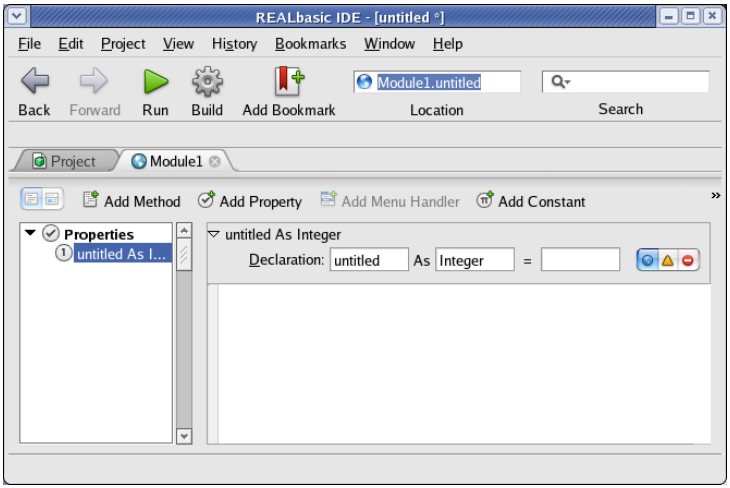
1 **If the Code Editor for the module is not already open, double-click the module’s name in the Project Editor to open it.**

The Module’s Code Editor appears.

2 **Click the Add Property button in the Code Editor toolbar or choose Project ▶ Add ▶ Property.**

The Property declaration area appears above the Code Editor area. A “placeholder” property declaration is entered by default.

Figure 245. The Property declaration area.



The Property Declaration area has three fields. They are for the name of the property, its data type, and its default value. The first two are required. If you do not provide a default value, the new property will take the default value for the data type that you choose. Strings have a default value of a empty string, numbers have a default value of zero, booleans have a default value of False, colors have a default value of black, and objects have a default value of Nil.

3 Fill in the Name and Data Type fields and, if desired, provide a default value.

A property can be an array. For example, if you want to declare a four-element integer array of properties called myProperties, you would write **myProperties(3)** in the Name field and **Integer** in the Data Type field.

You can also declare a property as an array with no elements and add elements later. In that case, you would declare myProperties using empty parentheses, **myProperties()**.

4 Choose a Scope for the property by clicking one of the three Scope buttons.

Your choices, from left to right, are Global, Public, and Protected. Note that elsewhere in REALbasic, the icons stand for Public, Protected, and Private.

Figure 246. The Scope buttons.

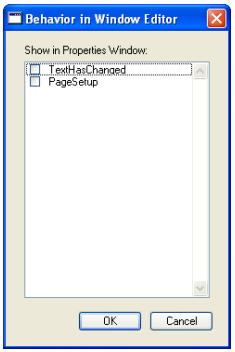
For information on Scope, see the section “Scope of a Module’s Methods, Properties, and Constants” on page 299.

If a property is Public or Protected, a Scope icon appears with a badge in the Code Editor browser.

5 **(Optional) If you want the property to be listed in the Properties pane in the Window Editor, right+click (Control-click on Macintosh) the name of any property in the browser and choose Window Editor Behavior.**

The Behavior in Window Editor dialog box appears. it lists all the properties that you have declared for the module.

Figure 247. The Behavior in Window Editor dialog box.



6 **Check each property that you want to have listed in the Properties pane.**

The "Show in Properties Window" option means that, if you drag an instance onto a window, the property can be assigned a value from the Properties pane (rather than only with code).

When you are finished, the property declaration appears in the Properties group in the browser area.

7 **(Optional) In the Code Editor area, add notes and comments about the property.**

The text entered into the Code Editor for a property is automatically non-executable, even if you write valid REALbasic code. Add any comments you wish, including code samples.

If you are creating a module for the sole purpose of adding properties to your application that will be global (accessible from everywhere in the application), consider placing them in the App class that was included in your project by default. Declare the Scope of these properties as Public. This enables you to access them throughout the application using the syntax. App.*PropertyName*.

They will still be global and this approach is more object-oriented since the properties are associated with the application directly rather than with a module that happens to be part of the application. See the section "The Application Class" on page 453 for more information on the App class.

Adding Constants to Modules

A constant acts like a variable but it holds a fixed value for its entire "life." When you create a constant, you give it its value. You can read the constant's value in your code, but you cannot use an assignment statement to change the value of a constant.

You can create constants in REALbasic for windows, modules, and classes that are added to the Project Editor. You can also create a local constant inside any method you write.

Each constant has a Scope. The Scope determines which parts of your application can "see" the constant and read its value. A constant added to a module can be Global, Public, or Protected in Scope. For information about Scope, see "Scope of a Module's Methods, Properties, and Constants" on page 299.

Global constants cannot be assigned non-printing characters such as Return, Tab, Space, and so on. One way to create a global object that returns a non-printing character is to add a function to a module that returns the desired character. For example, to create an object that returns a Carriage Return character (ASCII 13), add the following function to a module:

```
Function CR as String
  Return Chr(13)
```

Adding a Constant to a Module

To add a constant to a module, do this:

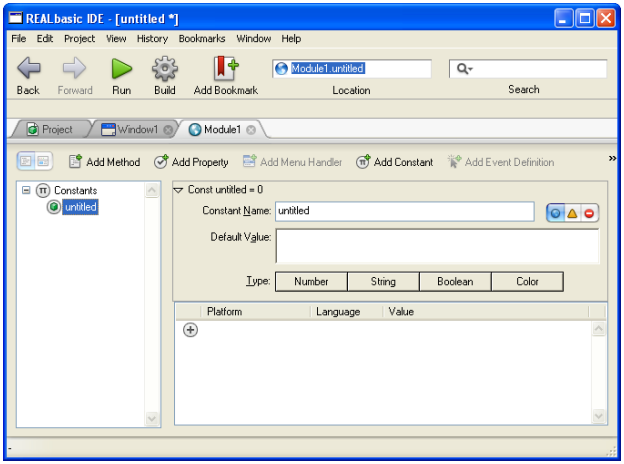
1. **If the Code Editor for the module is not already open, double-click the module's name in the Project Editor to open it.**

   The Code Editor for the module appears.

2. **Click the Add Constant button or choose Project ▶ Add ▶ Constant.**

   The Add Constant declaration area appears above the Code Editor area.

Figure 248. The Add Constant declaration area.



3 **Enter the name of the constant and its value.**

4 **Set the data type for the constant by clicking one of the four data type buttons, Number, String, Boolean, or Color.**

If you chose Color, a color patch will appear to the right of the Color button. Click it to display a Color Picker dialog. If the color patch does not appear, expand the IDE window so that there is space to the right of the Color button.

5 **Choose a Scope for the constant by clicking one of the three Scope buttons.**

Your choices, from left to right, are Global, Public, and Protected. Note that elsewhere in REALbasic, the three icons stand for Public, Protected, and Private.

Figure 249. The Scope buttons.

For information on Scope, see the section "Scope of a Module's Methods, Properties, and Constants" on page 299.

NOTE: The New Constant pane supports standard Cut, Copy, and Paste operations.

Color constants

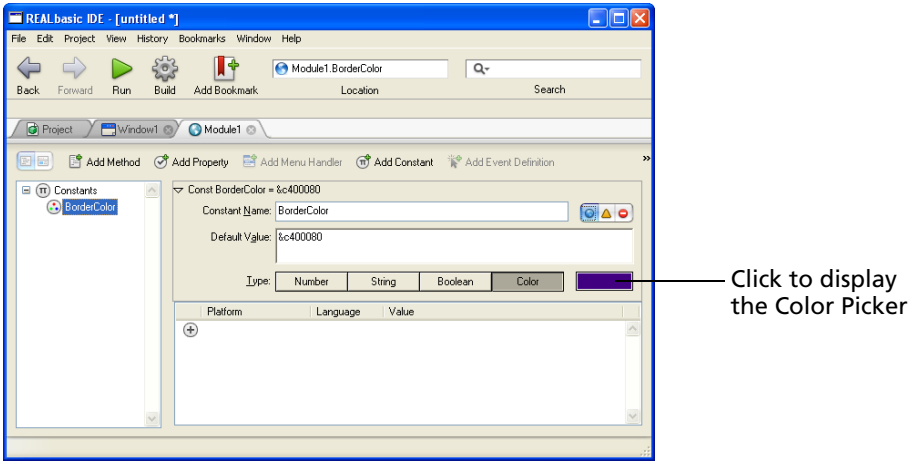
You can create constants of type color. Using color constants is a great way to ensure that the colors your application uses are consistent. When you specify the value of a color constant, you use the RGB (Red, Green, Blue) model. You specify the amounts of each primary color using the following format:

```
&cRRGGBB
```

where *RR* is the value of Red in hexadecimal, *GG* is the value of Green in hexadecimal, and *BB* is the value of Blue in hexadecimal. It's easy to obtain the desired values because the Add Constant declaration area includes a Color Picker

when you specify "Color" as the data type. Click on the color patch to the right of the value area to display the Color Picker and select the color. When you close the Color Picker, REALbasic inserts the RGB values for the selected color in the Value area. A color constant is shown in Figure 250.

Figure 250. A constant of type Color.



Using Constants to Localize your Application

Global constants also provide a very convenient way to localize your application. If you use global constants for all the text that appears in your application's interface, you can instantly localize the application simply by changing the Default Language setting in the Build Settings dialog box when you are ready to create a standalone application. For more information, see the section "Building Your Application" on page 548.

Public constants are equally suitable for localization; the only advantage of Global scope is that you can refer to them by name only, without needing to precede the name with the name of the module, class, or window in which they were declared. If you like, you can put your localization constants in the App class or even in a window, as long as you make them Public. The App class in the default Desktop Application project has constants that are used in the IDE to customized menu items on Macintosh, Windows, and Linux.

The Localization table at the bottom of the Add Constant declaration area lets you assign different values to the constant depending on platform and language. When you change the Default Language in Project Settings or the Build Application dialog box, the corresponding values for each constant take effect automatically. All you need to do is define a value for each combination of platform and language that you build.

Your choices for platform are:

- Windows (Windows 98/ME, NT, 2000, XP),

- Macintosh (any version of the Mac OS that supports REALbasic),
- Macintosh "classic" only (any Mac OS 'classic' version that supports REALbasic),
- Macintosh Carbon (PEF format; runs on both classic and OS X systems),
- Macintosh (Mach-O format; runs on Mac OS X only),
- Linux (GTK 2.x).

This set of features allows you to create different definitions for the constant for each type of operating environment.

Dynamic Constants

String constants that have values for multiple languages can be set to be dynamically localizable. That means that the built application will load the proper values for the language on which the application is running. On Mac OS X, it will generate .lproj folders inside of a package that will contain a single Localizable.strings file with the values of the strings. On Linux and Windows, it will generate .mo files according to the gettext format.

To enable the dynamically localizable feature, click the Dynamic checkbox that appears to the right of the blocks of data types. The Dynamic checkbox appears only if you have selected String as the constant's data type.

REAL Software provides a free localization utility, Lingua, that enables you to localize strings outside of the REALbasic IDE. To utilize Lingua, you first set up all of your string constants to be localized as "Dynamic."

An Example of a Localized String Constant

The following illustrates how to set up a constant that will be used as the caption for a button control that is used generically as the 'accept' button (i.e., "Save", "OK", and so forth). This illustrates how to localize within REALbasic; for information on how to localize externally, see the section "Using Lingua to Localize your Application" on page 311.

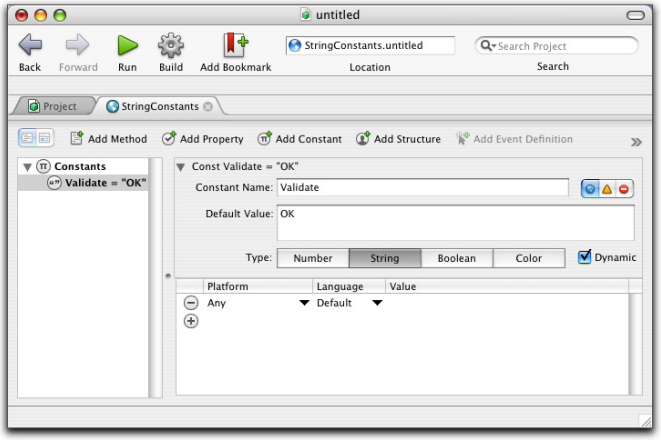
To define several values for a constant, do this:

1. **Using the New Constant declaration area, add a new constant.**
2. **Enter a value for the constant for the default language and set the type to String.**
3. **Choose the Scope for the constant.**

   Ordinarily you will make the scope of the constant Global so that it can be used in menus, menu items, and in windows and their controls.
4. **Click the Plus sign in the Localization table.**

   REALbasic adds a new blank row to the Localization table. This is shown in Figure 251.

Figure 251. A new row added to the Localization table.



The Platform column enables you to choose whether the value will be for any platform, any Macintosh platform, Macintosh 'carbon' (OS X and carbon classic), Mac OS X only (Mach-O), Mac 'classic' only, any Windows version, or Linux. The Language column offers choices of Default (the default language of the host OS) or virtually any specific language.

5 **Use the pop-up menus to choose a Platform and Language and enter the value for that platform/language combination in the Value area.**

If you are entering a string constant that will be used as an item's Text property and want to assign a keyboard accelerator, precede the letter with the ampersand (&) character. On Windows and Linux, the key will be underlined.

6 **Repeat these steps to add additional platform/language combinations, as needed.**

It is mandatory that a value be provided for each Platform/Language combination that you build. If you omit a value, REALbasic will have no idea what to use when you build the application for that platform and language.

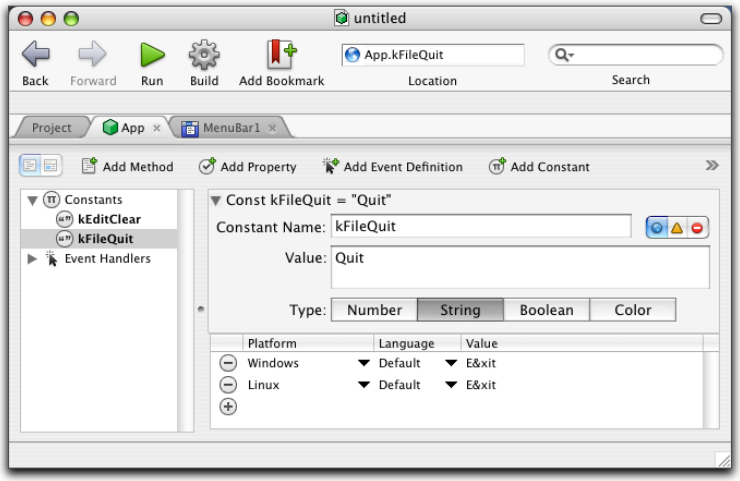
You can localize menus and menu items in exactly the same way. Create a global constant for each text string that will be used as a menu and menu item. Then use a constant's name as the menu's Text property, preceded by the number sign ("#"). If you want to specify a keyboard accelerator, place an ampersand prior to the accelerator character.

A Localization table example

The App class that is included in Desktop applications by default contains constants that are used to manage the File ▶ Exit and the Edit ▶ Delete menu items. On Macintosh, these menu items are Quit and Clear respectively. MenuBar1 contains references to these constants rather than literal text. Since the constant uses different values by platform, the localization table is filled in for platform only, not language.

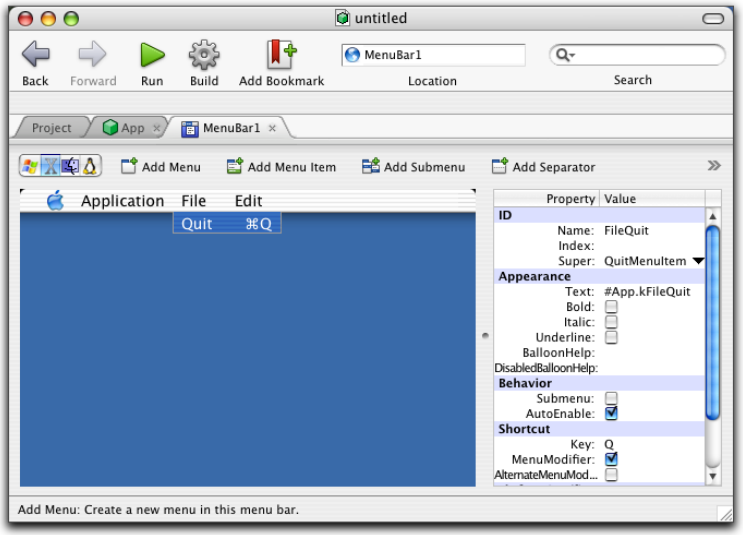
For example, the constant for the File ▶ Exit menu item specifies the value of "Quit" which is used on Macintosh, and has entries in the Localization table for Windows and Linux. They change the Value to "Exit" and specify a keyboard accelerator.

Figure 252. The specifications for the kFileQuit constant.



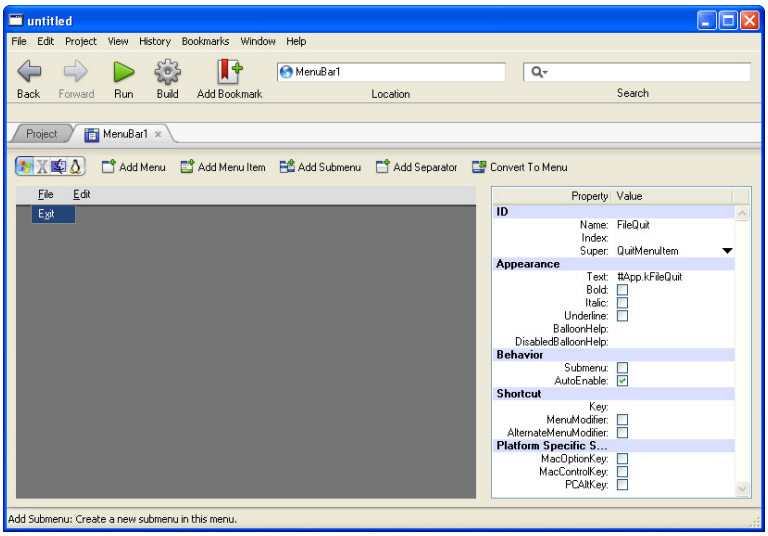
The kFileQuit constant is referred to in the Text property of the Quit menu item in the Menu Editor for MenuBar1. Note the use of the number sign in the Text property. Since the constant's Scope is Public, not Global, it is necessary to include the name of the object in which it is declared, thus it's "#App.kFileQuit". Notice that the Menu preview area shows that it has picked up the value of "Quit".

Figure 253. The properties of the Quit menu item (Macintosh).



On the Windows version of the same application template, the Menu Editor shows that the value of "Exit" is used.

Figure 254. The Exit menu item in the Windows IDE.



On Windows, the keyboard accelerator is indicated by the underscore in the Localization table.

This technique also works for all static text that appears in windows: PushButtons, bevel button menus, contextual menus, tab panel labels, etc. You can reference the constant simply by entering its name in the Properties pane, preceded by the number sign, as shown in Figure 254. If it is a public constant, you need to include the name of the object that contains the constant; if it is a global constant that was created in a module, then you can omit the name of the module that contains the constant.

You can also use constants in enterable fields in App class's Properties pane where you could also enter static text. For more information, see Chapter 14, "Building Stand-Alone Applications" on page 547.

Using Lingua to Localize your Application

REAL Software provides a free utility called Lingua that you can use to localize REALbasic applications. It does the localization outside the REALbasic IDE.

Lingua utilizes the dynamic constants option that you can select when you create a constant. The first step is to create dynamic constants for all strings that need to be localized. A recommended approach is to create a separate module for your localizable constants, but you can put these constants anywhere.

When all of your strings have been defined as dynamic constants, choose File ▶ Export Localizable Values. A dialog box will appear, asking you to select the

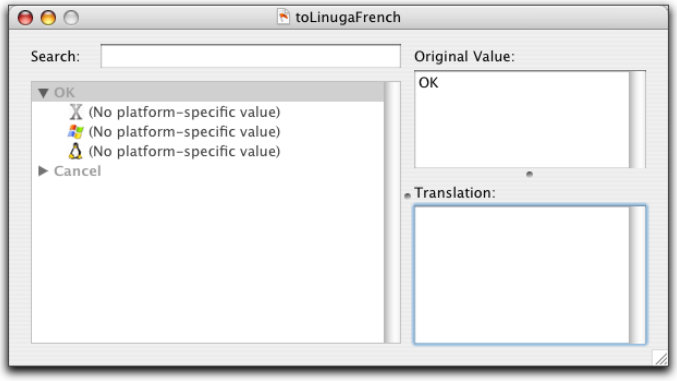
language you want to localize to. Select the language you are localizing to and click Export.

REALbasic will then write out a file that can be opened by Lingua on Windows, Linux, and Mac OS X.

Launch Lingua and then open the file you exported. The main Lingua window opens, showing a list of all the dynamic strings in your application. The values are grayed out when there is no localized version. If there are any different values specific to Windows, Linux, or Mac OS X, there will be an icon to the far right of the string in the list. Like the strings, the icons will be grayed out if the value is not localized.

To localize a string, select it in the list. The original value will be displayed in its entirety in the upper right pane, and in you can type the translated text in the lower right panel.

Figure 255. The Lingua main screen.



To add a value specific to a platform, expand the string in the list (as shown for the OK button, above), and select the individual platform to edit it.

To test the strings, choose File ⇣ Export to Application. Lingua presents an open-file dialog box. Select the target application and clic Open. When the import is complete, switch back to the REALbasic application and , debug the application normally.

When finished localizing, you can save the file from within Lingua and then import the strings file back into REALbasic by dragging it into a project or choosing File ▶ Import.

Structures

A *Structure* is a compound value type. It consists of a series of fields that are grouped together as a single block. You can control the size and order of the fields so you can declare a structure in REALbasic to match a structure defined by some external

library or as part of some binary file format or communications protocol. A structure can provide a convenient alternative to the MemoryBlock.

You might also use a Structure when porting a Visual Basic application to REALbasic; it is very similar in concept and syntax to Visual Basic's "User-Defined Type" feature, also known as a "UDT". In Visual Basic .NET this is called a *structure*.

A Structure is a data type, like an integer or a color. It is not a reference type like an object or an array. When you assign an object value to an object variable, you copy a reference to the object data; when you assign a structure value to a structure variable, you copy the entire contents of the structure. When you pass a structure as a parameter ByVal, the whole contents of the structure are copied; when you pass a structure ByRef, obviously the callee ends up modifying the caller's original structure instead.

The New operator does not apply to structures. When you create an array of structures, each element is an actual value (not a reference, like it would be with an array of objects). You can use the same dot syntax to access structure fields as you would use to access object properties, but when you use dot syntax with a structure, you are manipulating the structure variable itself, not a reference to data somewhere else.

Creating a Structure

Structures can be created only in modules. They can be given Global, Public, or Private scope. A structure contains a list of fields and/or arrays. You must declare the data type of each field or aray.

Structure fields can be defined as arrays, using the usual array syntax:
fieldName(UBound) As *DataType*

Arrays in structure fields can't be manipulated in the same ways as normal arrays; they represent a fixed chunk of storage inside the structure, not a dynamic object that can be resized and manipulated. Structure field arrays cannot be resized, cannot be assigned, and do not support any of the array methods.

Strings also have a special syntax and behavior inside a structure:

fieldName As String * size

A string in a structure is a simple array of bytes. It has a fixed size and does not store text encoding information. Just as you convert text to a specific encoding when writing it to a file or a socket, and assign the correct encoding to it when reading it back in, you must convert strings to a specific encoding when you assign them to a structure and define them to the correct encoding when reading them back out.

Structure fields can contain any of the simple value types, but cannot contain objects. The Structure Editor in the IDE will show you the size and location of each field, so that you can match your structure up exactly with an external data format.

To create a Structure, do this:

1 Open a module and choose Project ▸ Add ▸ Structure.

If you have added the Add Structure button to the Code Editor toolbar, you can click that button instead. A structure has a name and a list of fields, and can be global or private.

2 **In the structure declaration area, give the structure a name.**

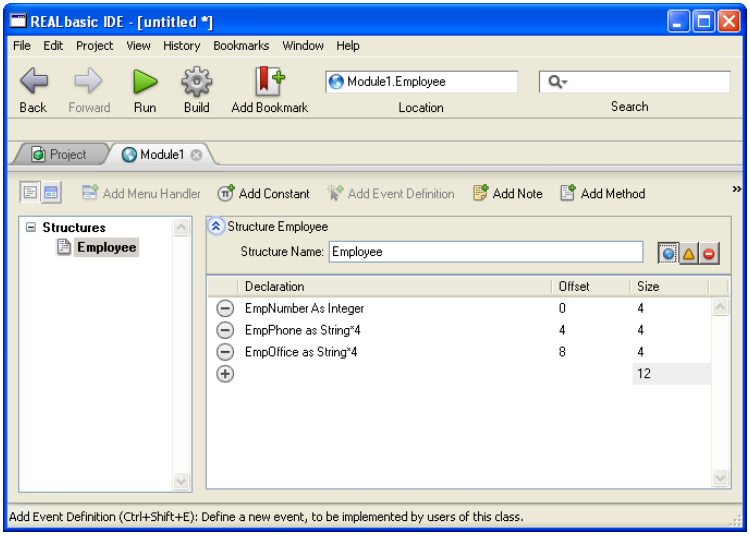
3 **Click the plus button in the field list to create a new field, then type in the declaration.**

The field declaration syntax is the same as for a Dim statement or a property declaration: *fieldName* As *DataType*

Like a property or local variable, fieldnames must be simple identifiers and must be unique within the field.

Here is a completed structure definition.

Figure 256. A Structure declaration.



Using Structures

Once you've defined a structure, you can use it in almost any context where you would use any other datatype. Use the dot syntax to access the fields. You can define an object or module property as a structure; you can declare a method parameter as a structure; you can even embed one structure as a field in another. Variants, however, cannot contain structures (try storing the StringValue instead), and you cannot return a structure from a method or event (try passing it as a ByRef parameter instead).

In addition to the fields you define, structures contain two built-in items:

Name	Parameters	Description
Size		This constant returns the total size of the structure in bytes.

Name	Parameters	Description
StringValue	littleEndian as Boolean	Lets you treat the structure as a string. This is useful for copying structures into and out of MemoryBlocks, for reading and writing structures to files, and for transmitting structures through sockets. You must pass the desired endianness, which should match the LittleEndian property of the MemoryBlock. The StringValue method of the BinaryStream class will convert the structure's fields to or from the appropriate endianness as necessary.

Enums

An *enum* or enumeration is a set of constants. It's a group of constants that are assigned values. You can assign a value to each constant or accept the default values. By default, the constans are numbered consecutively, starting with zero.

When you create an enum, you create a new data type. Enums accept only integer constants. When you want to get an enum, you need to cast it to an integer data type.

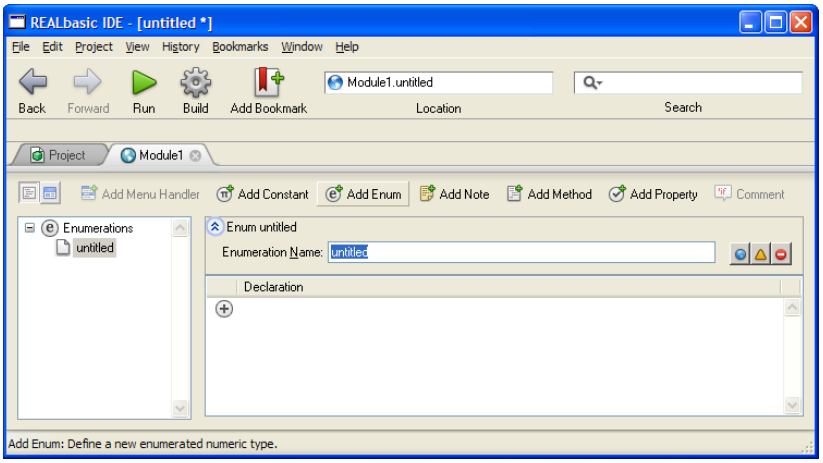
You can add an enum only to a module.

To create an Enum, do this:

1 **Open a module and choose Project ▸ Add ▸ Enum.**

If you have added the Add Enum button to the Code Editor toolbar, you can click that button instead. An Enum has a declaration area in which you name the Enum and set its Scope.

Figure 257. The Enum Declaration area.



2 **In the declaration area, give the Enum a name and click one of the Scope buttons to set its scope.**

3 **Click the plus button in the list to enter the name of the first constant.**

4 **If desired, assign a value to the constant by typing an equals sign, followed by the value.**

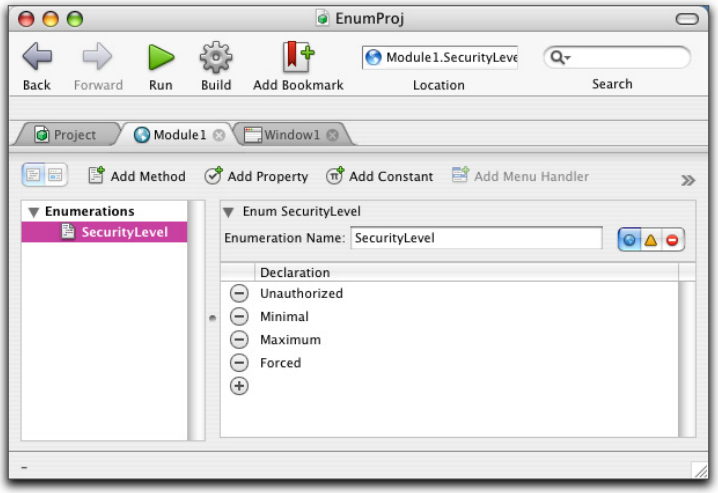
For example, if the first constant is named "Windows" and you want its value to be 13, you'd write:

```
Windows = 13
```

If you only enter the constant name, its value is its sequence number in the list, with the first item being zero.

Here is an example of a finished Enum. It defines the Enum named "SecurityLevel" and gives it four constants: Unauthorized, Minimal, Maximum, and Forced. Their values range from 0 to 3 since no values are included in the definition.

Figure 258. A global Enum defined for 'Security Level'.



You use the dot notation to get the values of the items. For example, the expression

```
SecurityLevel.Maximum
```

accesses the value of 2 because Maximum is the third constant in the Enum definition. To return the integer 2, you need to explicitly cast the enum using the desired integer data type. There is no implicit conversion from the enum data type to an integer data type. For example, the following code in a window returns the integer value 2 associated with this item.

```
Dim i as Integer
i=Int32(SecurityLevel.Maximum)
```

You can also declare a variable of the data type of the enum and get its values that way:

```
Dim MaxSec as SecurityLevel
Dim i as integer
MaxSec=SecurityLevel.Maximum
i=int32(MaxSec)
```

Importing and Exporting Modules

Modules can be imported from other REALbasic projects. Modules that have been exported from other projects appear on the desktop with a cube icon.

Importing and Exporting Modules

A nice feature of modules is that they are modular. For example, you might want to create a module that contains basic trig functions that you use in animations. You can easily reuse the module in other projects.

Exporting

Modules can be exported for use in other REALbasic projects. You can export a module using two different procedures:

- Drag the module from the Project Editor to the desktop.
- Click on the module in the Project Editor to select it and choose File ▶ Export Module.

You can use either procedure for encrypted modules. The first procedure is easier if you can see the folder, the desktop, or the disk you wish to copy the module to. If you need to drill down to a specific folder, use the second procedure.

Either procedure will export the module in its current state—protected or unprotected. Encrypted and unprotected modules share the same desktop icon; an encrypted module's protected status is apparent only within the Project Editor.

Figure 259. An exported module's desktop icon (Windows).

Modules can be encrypted prior to exporting to prevent other developers from accessing your code (see the following section, "Encrypting Modules" on page 318). When an encrypted module is imported into another project, it appears in the Project Editor with a key icon ( ); if the developer double-clicks the module to display its Code Editor, he is prompted to enter the decryption password.

The developer can *use* the encrypted module in the project, but he/she cannot access any of the module's code. Encryption is a great way to sell standalone modules that

provide enhancements to others' applications without the risk of having your work stolen.

Importing

To import a module into your project, drag the module into your Project Editor. Or, choose File ▶ Import and locate the module to be imported using the open-file dialog box. If the module is encrypted, the locked version of the module's icon appears in the Project Editor.

If you need to share a module among two or more projects, you can import it as an external project item. For more information, see the section "External Project Items" on page 56.

Encrypting Modules

You can encrypt (protect) or decrypt (unprotect) a module while it is in your project. When a module is encrypted, no one can access its code (including you) without supplying the decryption password.

When encrypting a module, you supply a password which can be used to decrypt it later.

To encrypt a module, do this:

1 **Right+click on the module in the Project Editor (Control-click on Macintosh) and choose Encrypt from the contextual menu or choose Edit ▶ Encrypt.**

You can optionally add an Encrypt button to the Project Editor toolbar. If it is available, you can encrypt a module by selecting the module in the Project Editor and clicking the Encrypt button.
The Encrypt *Module* dialog box appears, as shown in Figure 260.

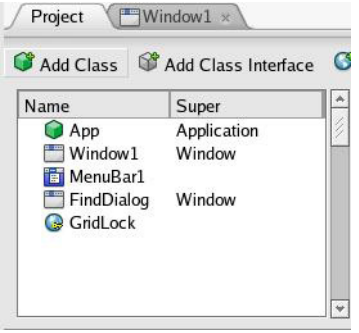
Figure 260. The Encrypt *Module* Dialog box.

2 **Enter and confirm a password for encryption.**

Important Note: Don't forget your password.

An encrypted module appears in the Project Editor with a small key in the lower right corner of the module icon. This is shown in Figure 261.

Figure 261. A project with an encrypted module.



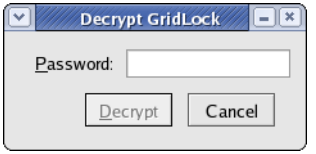
When a programmer tries to open an encrypted module, REALbasic presents the Decrypt *Module* dialog box, shown in Figure 262.

To decrypt an encrypted module, do this:

1 **Right+click on the module in the Project Editor (Control-click on Macintosh) and choose Decrypt from the contextual menu or choose Edit ▸ Decrypt.**

The Decrypt *Module* dialog box appears.

Figure 262. The Decrypt *Module* dialog box.



2 **Enter the decryption password and click Decrypt.**

If the correct password was entered, the key will disappear from the module's icon, indicating that it has been successfully decrypted. If you entered an incorrect password, a message box will inform you of that fact.

CHAPTER 7

Working With Text and Graphics

Almost every application manipulates text and graphics in some way. Fortunately, REALbasic provides a rich set of functions for creating, manipulating, displaying, and printing text and graphics. Should you wish to create your own custom control, you can use the Canvas control and its graphics methods to create it.

Contents

- Creating Animation with Sprites
- Creating 3D animation with the Rb3DSpace control

Working With Fonts

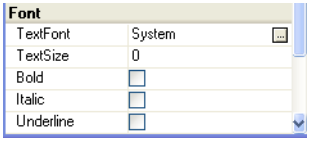
REALbasic gives you the ability to set the font, font size, and font style of many of the objects and controls in your application. EditFields support multiple fonts, styles, and sizes (collectively referred to as *styled text*) and ListBoxes support multiple styles. Controls that use a single font have a TextFont property that you can set by assigning it the name of the font you want used to display text for the control. EditFields have a TextFont property but they can also display multiple fonts. For information on styled text in EditFields, See "Handling Styled Text" on page 326.

The System and SmallSystem Fonts

The System font is the font used by the system software as its default font. It's the font used for the menus as well.

If you want text to be displayed or printed in the user's System font, use the name "System" as the font when you assign it. You can enter it as the TextFont property in the Properties pane. If you also enter zero as the TextSize, REALbasic will choose the font size that works best for the platform on which the application is running. Because of differences in screen resolution, different font sizes are often required for each platform. This feature enables you to use different font sizes on different platforms without having to create separate windows for each platform. Use the Properties pane to set the TextFont to "System" and the TextSize to zero.

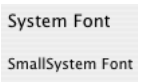
Figure 263. Using the System font with Font Size of 0.



If the system software supports both a large and small System font, you can also specify the "SmallSystem" font as your TextFont. This option selects the small system font on the user's computer, if there is one. If there is no small system font, the System font is used.

On Mac OS X, both System font and the small System font are supported. Figure 264 illustrates the difference between the two fonts. The TextFont is "SmallSystem" and the TextSize is zero in both cases.

Figure 264. The System and SmallSystem fonts on Mac OS X.



What Fonts Are Available?

You may want to use fonts other than the System font. In this case you will need to determine if a particular font is installed on the user's computer. REALbasic has two global functions, FontCount and Font, that make determining available fonts easy. The following function, when passed a font name, will return True or False to inform you if the font passed is installed:

```
Function FontAvailable(FontName as String) As Boolean
Dim i,nFonts as Integer
nFonts=FontCount-1
For i=0 to nFonts
  If Font(i)=FontName Then
    Return True
  End If
Next
Return False
```

The following code can be used in the Open event handler of a PopupMenu or ListBox to build a list of all available fonts:

```
Dim i,nFonts as Integer
nFonts=FontCount-1
For i=0 to nFonts
  Me.AddRow Font(i)
Next
```

To add a Font menu to your application, do this:

1. **Add a menu with the name "Font" and set the Text property to "Font".**
2. **Add an item to the Font menu set the Text property to "FontName". REALbasic will automatically name the new item "FontFontName".**
3. **Set the Index property of the menu item to 0 (zero).**
4. **Put the following code in the Open event handler of the App class:**

```
Dim m as MenuItem
Dim i,nFonts as Integer
nFonts=FontCount-1
FontFontName(0).text=Font(0)
For i=1 to nFonts
  m=New FontFontName
  m.text=font(i)
Next
```

All of these menu items will share one menu handler. This menu handler will be passed an Index parameter which will indicate which menu item is passed. This Index parameter can be used in conjunction with the Font function to determine which font was selected.

In the REALbasic Tutorial, you build a Font menu that uses a BevelButton control rather than a menu in the menu bar. The Font menu is built using the AddRow method of the BevelButton control:

```
Dim i, nFonts as Integer
nFonts=FontCount-1
For i=0 to nFonts
  me.AddRow Font(i)
Next
me.Caption=TextField.SelTextFont
```

The last line of the method tells the BevelButton to display the current font used in the text editor application.

You can use the same programming logic to build a Font menu using a PopupMenu control; it also has an AddRow method.

Working with the Selected Text

The "Selected Text" refers to text that is selected (or "highlighted") in the EditField that currently has the focus. EditFields have three properties that can be used to get and/or set the selected text.

Table 14: Properties for getting or setting selected text.

Name	Description
SelLength	The number of characters currently selected. You can change the selected text by changing this number. Setting this value to 0 (zero) will position the insertion point based on the value in the SelStart property rather than selecting any text.
SelStart	The number of the character just before the selected text. For example, if the fifth character in an EditField was selected, this property would be 4. Setting this value to 0 (zero) will start the selection at the beginning of the EditField.
SelText	A string containing all of the selected text. Changing this value will replace the selected text with the SelText value. If no text is selected, the SelText value will be inserted at the insertion point (the value in SelStart).

This code selects all the text in EditField1:

```
EditField1.SelStart=0
EditField1.SelLength=Len(EditField1.Text)
```

If you need to execute some code when the user moves the insertion point or highlights some characters, place your code in the SelChange event handler of the EditField.

Creating a Password Field

EditFields have Password and LimitText properties that can be used to create password fields. When you set the Password property, asterisks (on Windows and Linux) or bullet characters (on Macintosh) appear instead of the characters you type. However, the characters you enter are placed in the EditField's Text property. The LimitText property allows you to control the maximum number of characters the user can type in the EditField.

NOTE: The Password property will function only if the MultiLine property is False (not checked).

Formatting and Filtering Text Entry

The EditField has two properties that enable you to format text when it is entered and filter entries on a character-by-character basis. These properties are especially useful when an EditField is used as a data entry field in a database or has an equivalent role in applications that don't explicitly use a database engine to store the information.

For example, if an EditField is used to enter a US Social Security number, a valid entry must adhere to a specific format—namely three numbers followed by a dash, two more numbers and another dash, and four numbers, e.g., "578-68-7891". Many other types of information follow a regular structure—phone numbers, credit card numbers, drivers licence codes, and so forth.

The Format Property

The EditField's Format property is designed to allow an EditField to have a different display value than what was entered. For example, you could display all numbers with 2 digits of precision using this property. When the EditField has the focus, any formatting is cleared and the text is shown in its unaltered state. If an EditField doesn't have the focus, and you have specified a format, the text will be shown according to the given format. The formats supported are exactly the same as those supported in the Format function. See the Format function in the *Language Reference* for definitions.

To turn off formatting, set the format property to the empty string, "" — two quotes with nothing between them.

Example Formats

Here are some sample formats for real numbers:

```
// Always display number with 2 decimal places pad with 0 if necessary.
EditField1.format = "0.00"

// Display at most 2 decimal places but don't pad with 0.
EditField1.format = "#.##"

// Turn off formatting
EditField1.format = ""
```

The Mask Property

The EditField's Mask property is designed to permit only certain types of characters in certain positions of the field. For example, if you want users to enter a US Social Security number, you can use a mask to ensure that the user doesn't type any letters (numbers only), and that the number given is only nine digits long. Each character in the mask corresponds to a place holder for a specific type of character or a literal character. Such a mask is "###-##-####".

The input mask is completely compatible with Visual Basic with the exception of the "~" which REALbasic reserves for future use and expansion.

If an EditField has a Mask property, there is no visible indication of that to the end user until text that is not permitted by the Mask is rejected or the user tries to enter too many characters.

See the entry for EditField in the *Language Reference* for the symbols you can use in a mask.

Example Masks

Here are some simple masks that illustrate how the feature works.

```
// Allow U.S. Social Security numbers to be entered.
EditField1.Mask = "###-##-####"

// Allow dates of the form 14-Dec-1972
EditField1.Mask = "##-???-####"

// Auto-capitalize a serial number of the form AWS-1925-ASD
EditField1.Mask = ">???-####-???"

// Turn off the mask
EditField1.Mask = ""
```

Handling Styled Text

The term *styled text* means text that can have more than one font, font size, and/or font style. In order for an EditField to display styled text, its MultiLine property must be True (checked) and its Styled property must be True (checked). In order to

print styled text, you must use the StyledTextPrinter class. See the section "Printing Styled Text" on page 364 for more information.

Determining the Font, Size, and Style of Text

EditFields have properties that make it easy to determine the font, font size, and font style of the selected text in an EditField. The SelTextFont property can be used to determine the font of the selected text. If the selected text has only one font, the SelTextFont property contains the name of that font. If the selected text uses more than one font, the SelTextFont property is empty.

This function returns the names of fonts for the selected text of the EditField passed:

```
Function Fonts(item as EditField) as String
  Dim fonts, theFont as String
  Dim i, Start, Length as Integer
  If Field.SelTextFont="" Then
    Start=Field.SelStart
    Length=Field.SelLength
    For i=Start to Start+Length
      Field.SelStart=i
      Field.SelLength=1
      If InStr(fonts,Field.SelTextFont)=0 Then
        If fonts="" Then
          fonts=Field.SelTextFont
        Else
          fonts=fonts+", "+Field.SelTextFont
        End if
      End if
    Next
    Return fonts
  Else
    Return Field.SelTextFont
  End If
```

The SelTextSize property is used to determine the font size of the selected text and works the same way as the SelTextFont property. If all characters of the selected text are the same font size, the SelTextSize property will contain that size. If different sizes are used, the SelTextSize property will be 0.

There are also boolean properties for determining if all of the characters in the selected text are the same font style. Since text can have multiple styles applied to it, these properties determine if all of the characters in the selected text have a particular font style applied to them. For example, if all of the characters in the selected text are bold but some are also italic, a test for bold returns True. On the other hand, a test for italic returns False since some of the selected text is not in the italic font style. For all of these properties, you test to see if the property is True or False. If the test returns True, then all of the characters in the selected text have that font style. If it returns False, the selected text contains more than one font style. If you want to determine which styles are in use, you can programmatically select each

character in the selected text and then test the style properties. This is an operation similar to the sample Fonts function that determines which fonts are in use in the selected text. The properties for testing the various available font styles are:

Table 15: Properties that test for font styles[a].

Property	Style
SelBold	Bold
SelItalic	Italic
SelUnderline	Underline
SelOutline	Outline
SelShadow	Shadow
SelCondense	Condensed
SelExtend	Extended

a. The Outline, Shadow, Condensed, and Extend styles are supported only on Mac OS "classic."

In this example, if the selected text of the EditField is bold, then the Bold menu item is checked:

```
StyleBold.Checked=EditField1.SelBold
```

If all of the characters in the selected text are not bold then EditField1.SelBold returns False which will then be assigned to the Checked property of the StyleBold menu item.

Setting the Font, Size, and Style of Text

The properties used to check the font, font size, and font styles of the selected text are also used to set these values. For example, to set the font of the selected text to Helvetica, you do the following:

```
EditField1.SelTextFont="Helvetica"
```

Keep in mind when setting fonts that the font must be installed on the user's computer or the assignment will have no effect. You can use the FontAvailable function mentioned earlier in this chapter to determine if a particular font is installed.

You can set the TextSize property of a control to zero to tell REALbasic to use the font size that looks best for the platform on which the application is running.

You can set the font size of the selected text using the SelTextSize property. For example, the following code sets the font size of EditField1 to 12 point:

```
EditField1.SelTextSize=12
```

To apply a particular font style to the selected text, set the appropriate style property to True. For example, the following code applies the Bold style to the selected text in EditField1:

```
EditField1.SelBold=True
```

Table 15 on page 328 lists all the font style properties of EditFields that can be used in this same way.

EditFields also have built-in methods for toggling the font styles on and off. "Toggling" in this case means applying the style if some of the selected text doesn't have the style already applied or removing the style from any of the selected text that already has it applied. The following code toggles the bold style of the selected text in EditField:

```
EditField1.ToggleSelectionBold
```

The methods for toggling the styles of the selected text are shown in Table 16 on page 329.

Table 16: Methods for toggling text styles.

Method Name	Style[a]
ToggleSelectionBold	Bold
ToggleSelectionItalic	Italic
ToggleSelectionUnderline	Underline
ToggleSelectionOutline	Outline
ToggleSelectionShadow	Shadow
ToggleSelectionCondense	Condensed
ToggleSelectionExtend	Extended

a. Outline, Shadow, Condensed, and Extend are supported only on Mac OS "classic."

Working with StyledText Objects

When you are working with styled text that is displayed in an EditField, you can work with the properties of EditFields that get and set style attributes (as described in the previous section) to manage styled text. However, REALbasic also provides tools for opening, saving, and managing styled text separately from an EditField or any other control. In fact, the styled text doesn't even have to be displayed at all.

This set of techniques uses the properties and methods of the StyledText class. Its Text property contains the styled text that is managed by the StyledText object. It has six properties for getting and setting style attributes:

Table 17: Properties for getting or setting Style Attributes.

Name	Description
Bold	Gets or sets the Bold style to the selected text in *Text*.
Font	Gets or sets the font for the selected text in *Text*.
Italic	Gets or sets the Italic style to the selected text in *Text*.
Size	Gets or sets the font size to the selected text in *Text*.
TextColor	Gets or sets the color of the selected text in *Text*.
Underline	Gets or sets the Underline style to the selected text in *Text*.

Each method takes parameters for the starting position and length of the text for which the attribute applies. These numbers are zero-based. For example, a call to the Bold property would look like this:

```
Dim st as New StyledText
st.Text="How now Brown Cow."
st.Bold(0,3)=True
```

This sets the first word, "How", in bold. Each contiguous set of characters that has the identical set of style attributes makes up a *StyleRun* object. In this example, the first three characters make up one StyleRun. The remaining text is the second StyleRun. In the language of a word processor, each StyleRun is an instance of a character style. The entire Text property is made up of a sequence of StyleRuns.

The StyledText class has six methods for managing StyleRuns.

Table 18: Methods of the StyledText class for working with StyleRuns.

Name	Description
AppendStyleRun	Appends a StyleRun to the end of *Text*.
InsertStyleRun	Inserts a StyleRun at a specified position.
RemoveStyleRun	Removes a specified StyleRun from *Text*.
StyleRun	Provides access to a particular StyleRun in *Text*. The StyleRun class has its own properties that describe the style that's applied to all the characters in the StyleRun.
StyleRunCount	Returns the number of StyleRuns that make up *Text*.
StyleRunRange	Accesses the starting position, length, and end position of the StyleRun.
Text	The text that is managed by the StyledText object. Technically, Text is a method, but you can get and set its value as if it were a property.

The Text method of a StyledText object can have multiple paragraphs. A paragraph is the text between two end-of-line characters. A paragraph can be defined either

with the EndOfLine function or the end-of-line character for the platform the application is running on.

A paragraph can be made up of multiple StyleRuns. It has only one style property of its own, paragraph alignment (Left, Centered, or Right).

There are three methods of the StyledText class for working with paragraphs:

Table 19: Methods of the StyledText class for working with Paragraphs.

Name	Description
Paragraph	Provides access to a particular Paragraph in *Text*. The Paragraph class has its own properties that return the start position, length, end position, and alignment of the paragraph.
ParagraphCount	Returns the number of Paragraphs that make up *Text*.
ParagraphAlignment	Sets the alignment of the specified paragraph (Default, Left, Centered, or Right). The ParagraphAlignment method takes one parameter, the number of the paragraph to be aligned (starting at zero). You assign it a Paragraph alignment constant. The four alignment constants are: AlignDefault (0): Default alignment AlignLeft (1): Left aligned AlignCenter (2): Centered AlignRight (3): Right aligned For example, to right align the first paragraph, you would use a statement such as StyledText1.ParagraphAlignment(0)=Paragraph.AlignRight

Although you can work with a StyledText object entirely in code—without ever displaying it—the EditField control is "hooked up" to the StyledText class in the sense that you can access all the methods and properties of the StyledText class via the StyledText property of the EditField. In order to work with a StyledText object in an EditField, you must turn on the MultiLine and Styled properties of the EditField. You can do this using the Properties pane.

For example, the line:

```
EditField1.StyledText.Text="Here is my styled text."+EndOfLine _
        +"Aren't you really impressed?"
```

sets the Text property of the StyledText object and displays it in the EditField. From there, you can go ahead and assign style properties to the text. Here is a simple example that works with these two paragraphs:

```
Dim st,ln as Integer
Dim Text as String
Text="Here is my styled text."+EndOfLine+"Aren't you really impressed?"
EditField1.StyledText.Text=Text

//assign Font and  Size to entire text
EditField1.StyledText.Font(0,Len(Text))="Arial"
EditField1.StyledText.Size(0,Len(Text))=14

//apply character hightlights to 'my' in first paragraph
EditField1.StyledText.Bold(8,2)=True
EditField1.StyledText.textColor(8,2)=&cFF0000 //Red

//get positions of second paragraph; the index is zero-based.
st=EditField1.StyledText.Paragraph(1).StartPos-1
ln=EditField1.StyledText.Paragraph(1).Length

//Second paragraph in Bold
EditField1.StyledText.Bold(st,ln)=True
//Second paragraph Centered
EditField1.StyledText.ParagraphAlignment(1)=Paragraph.AiignCenter
```

The result is shown in Figure 265.

Figure 265. The StyledText as it appears in an EditField.

This example happens to work with the StyledText object 'hooked up' to the EditField, but you can also work with styled text 'offline'. You declare a StyledText object in a Dim statement and operate on it without reference to any control. When you're ready to display it, you can assign it to the StyledText property of an EditField. You would do this with a line such as:

```
Dim st As New StyledText
//do whatever you want right here; when you're done, just write...
EditField1.StyledText=st
```

You can also export the styled text as a series of StyleRuns and read them back in and reconstruct the StyledText object using the AppendStyleRun method. See the

entries on StyleRun and StyledText in the Language Reference for more information.

Working with Text Encodings

All computers use encoding systems to store character strings as a series of bytes. The oldest and most familiar encoding scheme is the ASCII encoding. It is documented in the *Language Reference*. It defines character codes only for values 0-127. These values include only the upper and lowercase English alphabet, numbers, some symbols, and invisible control codes used in early computers. You can use the Chr function to get the character that corresponds to a particular ASCII code.

Many extensions to ASCII have been introduced which handle additional symbols, accented characters, non-Roman alphabets, and so forth. In particular, the Unicode encoding is designed to handle any language and a mixture of languages in the same string. REALbasic supports two different Unicode formats, UTF-8 and UTF-16. All of your constants, string literals, and so forth are stored internally using UTF-8 encoding.

If the strings you work with are created, saved, and read within REALbasic, you shouldn't have to worry about encoding issues because REALbasic stores the encoding it uses along with the content of the string. This is also true of the REALdatabase engine. VarChar and String fields store the encoding of the text along with the text itself and use the stored encoding when the data is retrieved.

If you are creating applications that open, create, or modify text files that are created outside of REALbasic, you need to understand how text encodings work and what changes you may need to make to your code to make sure it continues to work properly.

Text Encodings: From ASCII to Unicode

As you know, computers don't really store or understand characters. They store each character as a numeric code. For example, the Return is ASCII character number 13. When the computer industry was in its infancy, each computer maker came up with their own numbering scheme. A numbering scheme is sometimes called a *character set*. It is a mapping of letters, numbers, symbols, and invisible codes (like the carriage return or line feed) to numbers. With a character set, information can be exchanged between computers made by different manufacturers.

In 1963 the American Standards Association (which later changed its name to the American National Standards Institute) announced the American Standard Code for Information Interchange (ASCII) which was based on the character set available on an English language typewriter.

Over the years, computers became more and more popular outside of the United States and ASCII started to show its weaknesses. The ASCII character set defines only 128 characters. That covers what is available on an English-language typewriter, plus some special "control" characters that can be used on computers to control output. It doesn't include special characters that are commonly used in

typeset books such as curved quotes or the curved apostrophe, bullet characters, and long dashes—like this one. Also, many languages (like French and German) use accented characters that are not defined as part of the ASCII specification.

When the Macintosh and Windows operating systems were introduced, each OS defined extensions to standard ASCII by defining codes from 128-255. This enabled both operating systems to handle accented characters and other symbols that are not supported by the ASCII standard. However, the Macintosh and Windows extensions do not agree with one another. Cross-platform applications have to build in some way of managing text that uses characters in the 128-255 range.

The problem is even worse for users of languages that don't use the standard Roman alphabetic characters at all—like Japanese, Chinese, or Hebrew. Because there are so many characters, the character sets devised to support some of these languages use two bytes of data per character (rather than one byte per character, as in ASCII).

Apple eventually created various text encodings to make it easier to manage data. MacRoman is a text encoding for files that use ASCII. MacJapanese is a text encoding for files that store Japanese characters. There are others as well. But these encodings were Mac specific. They didn't make exchanging data with other operating systems any easier and mixing data with different encodings (typing a sentence in Japanese in the middle of an English-Language document, for example) was problematic.

In 1986, people working at Xerox and Apple Computer both had different problems to solve that required the same solution. Before long, the concept of a universal character encoding that contained all the characters for all languages, became the obvious solution. The universal encoding was dubbed "Unicode" by one of the people at Xerox that helped to create it. Unicode solves all of these problems. Any character you need from any language is supported and will be the same character on any computer that supports Unicode. And as a bonus, you can mix characters from different languages together in one document since all are defined in Unicode.

Unicode support began appearing on the Mac with System 7.6 and on Windows with Windows 95. You could translate files between other text encodings and Unicode but Unicode was still the exception and not the rule. It wasn't until Mac OS X and Windows 2000 that Unicode became the standard.

Computer users are now in a transition. There are some using older systems where Unicode is not the standard. All new systems that are running Mac OS X, Windows, or Linux use Unicode as the standard encoding. As a result, you may have to deal with text files of different encodings for a while. That means you may need to modify your code to handle this. At some point in the future, it may be so rare that you can assume all files are in Unicode format but until then, you may need to make some modifications to your code so that your application operates properly when it encounters text with different types of encoding.

Changing Your Code To Handle Text Encodings

Unfortunately, there is no perfectly accurate way to determine the encoding of a file. You have to know what encoding the file is using. If it's coming from an English-speaking user of Mac OS 9 (for example) you can probably assume it is MacRoman (but it never hurts to ask). If it is coming from an English-speaking user of Windows 98, it's probably Windows ANSI.

If the encoding of a string is defined, you can use the Encoding function to get its encoding, like this:

```
TextEnc=Encoding(s)
```

where the variable s contains the string whose encoding is to be determined and TextEnc is a TextEncoding object. If the encoding is not defined, the Encoding function returns Nil.

Reading a Text File

This example code reads data from a text file and displays it in an EditField. It makes no assumptions about encoding. If it's not a Unicode file, it may not display properly. This example has been kept simple intentionally to focus on the encoding issue:

```
Dim f As FolderItem
Dim t as TextInputStream
f = GetFolderItem("Sample.txt")
t = f.OpenAsTextFile
EditField1.text = t.ReadAll
t.close
```

If you know the encoding, this code can easily be changed to read the file properly. The TextInputStream class has an Encoding property that you can use to set the Encoding of the text that will be read using either the Read or ReadAll methods.

In the following example, the Encoding property of the TextInputStream is set to MacRoman. Since the example file is in MacRoman format, it displays properly.

```
Dim f As FolderItem
Dim t as TextInputStream
f = GetFolderItem("Sample.txt")
If f<> Nil Then
  t = f.OpenAsTextFile
  t.Encoding = Encodings.MacRoman
  EditField1.text = t.ReadAll
  t.close
End if
```

The line that sets the value of the Encoding property calls the Encodings object. The Encodings object contains functions for all the different encodings. The default encoding is UTF-8, a specific format of Unicode. When you type "Encodings." in

the Code Editor and press Tab, the autocomplete feature of the Code Editor will display all the encodings that are available.

Instead of assigning a TextEncoding to the Encoding property, you can pass it as an optional parameter to the Read or ReadAll methods. Here is an example of this:

```
Dim f As FolderItem
Dim t as TextInputStream
f = GetFolderItem("Sample.txt")
If f<> Nil Then
  t = f.OpenAsTextFile
  EditField1.text = t.ReadAll( Encodings.MacRoman)
  t.close
End if
```

Writing a Text File

If your application needs to write out a file in a particular encoding, you must specify it when you write out data. This can come up when the file will be read by a different application, sent to someone in another country, or to someone using a different operating system.

You use the ConvertEncoding function to convert text in one encoding to another encoding. Here's a simple example that writes text from an EditField to a file specifying MacRoman as the encoding. The ConvertEncoding function converts the encoding of the text to MacRoman before passing it to the Write method:

```
Dim f As FolderItem
Dim t as TextOutputStream
f = GetFolderItem("Sample.txt")
If f<> Nil Then
  t = f.CreateTextFile
  t.Write ConvertEncoding(EditField1.text, Encodings.MacRoman)
  t.close
End if
```

If your application reads and writes its own files, you don't have to worry about this issue. UTF-8 is assumed when reading text files and is assumed when writing text to a file. If you do nothing, your files will be read as Unicode and will write out in Unicode format.

The same is true when working with the REALdatabase. It is "encoding-savvy" in the sense that it stores the encoding of text that it stores in VarChar fields. As long as you store and retrieve data from the REALdatabase, you don't need to worry about encoding issues.

Getting an Individual Character

As was mentioned earlier, when you need to obtain an individual ASCII character, you can use the Chr function by passing it the ASCII code for the character you want. But if you want a non-ASCII character, you must specify the encoding as well. The Chr function is for the ASCII encoding only; you may not get the expected

character if you pass it a number higher than 127. You should instead use the Chr method of the TextEncoding class. It requires that you specify both the encoding and the character value. For example, the following returns the ™ symbol in the variable, s:

```
Dim s as String
s=Encodings.MacRoman.Chr(170)
```

Formatting Numbers, Dates, and Times

REALbasic provides the ability to display and print numbers, dates, and times in many different formats.

Numbers

Numbers are stored unformatted. Fortunately, REALbasic provides a Format function that makes providing formatting to numbers easy. To use this function, pass it a format specification and the number you wish formatted. The Format function then returns a string that represents the number with the formatting applied to it. The syntax for the Format function is:

result=Format(*Number*, *FormatSpec*)

The FormatSpec is a string made up of one or more characters that control how the number will be formatted. For example, the format spec "$###,##0.00" applies the typical dollars and cents formatting used in the United States.

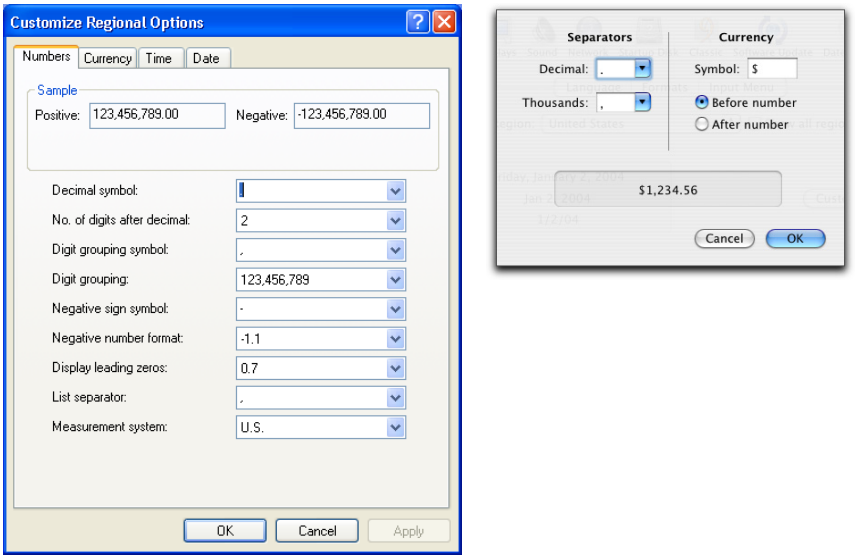
Table 20: Characters used in FormatSpec.

Character	Description
#	Placeholder that displays the digit from the value if it's present.
0	Placeholder that displays the digit from the value if it's present. If no digit is present, 0 (zero) is displayed in its place.
.	Placeholder for the position of the decimal point.
,	Placeholder that indicates that the number should be formatted with thousands separators.
%	Displays the number multiplied by 100.
(	Displays an open paren.
)	Displays a closing paren.
+	Displays a plus sign to the left of the number if the number is positive or a minus sign if the number is negative.
_	Displays a minus sign to the left of the number if the number is negative. There is no effect for positive numbers.
E or e	Displays the number in scientific notation.
\character	Displays the character that follows the backslash.

On Windows, the character that is used as the Decimal and Thousands separator is specified by the user in the Regional Settings Control Panel. In Mac OS X, these

characters are specified on the Numbers panel of the International system preference.

Figure 266. The Regional Settings and Numbers Control Panels.



By default, the FormatSpec applies to all numbers. If you want to specify different FormatSpecs for postive numbers, negative numbers, and zero, simply separate the formats with semi-colons within the FormatSpec. The order in which you supply FormatSpecs is: *positive*, *negative*, *zero*. The last three examples in Table 21 on page 338 show this. It shows some examples of FormatSpecs:

Table 21: Examples of various FormatSpecs.

Format Syntax	Result
Format(1.784, "#.##")	1.78
Format(1.3, "#.0000")	1.3000
Format(5, "0000")	0005
Format(.25, "#%")	25%
Format(145678.5, "#.##")	145,678.5
Format(145678.5, "#.##e+")	146e+5
Format(-3.7, "-#.##")	-3.7
Format(3.7, "+#.##")	+3.7
Format(3.7, "#.##; (#.##); \z\e\r\o")	3.7
Format(-3.7, "#.##; (#.##); \z\e\r\o")	(3.7)
Format(0, "#.##; (#.##); \z\e\r\o")	zero

Dates

Dates are objects and have properties that hold the date in various different formats. To get a date as a string formatted in a specific way, you simply access the appropriate property. Table 22 on page 339 lists the properties of date objects and an example of the format the property contains:

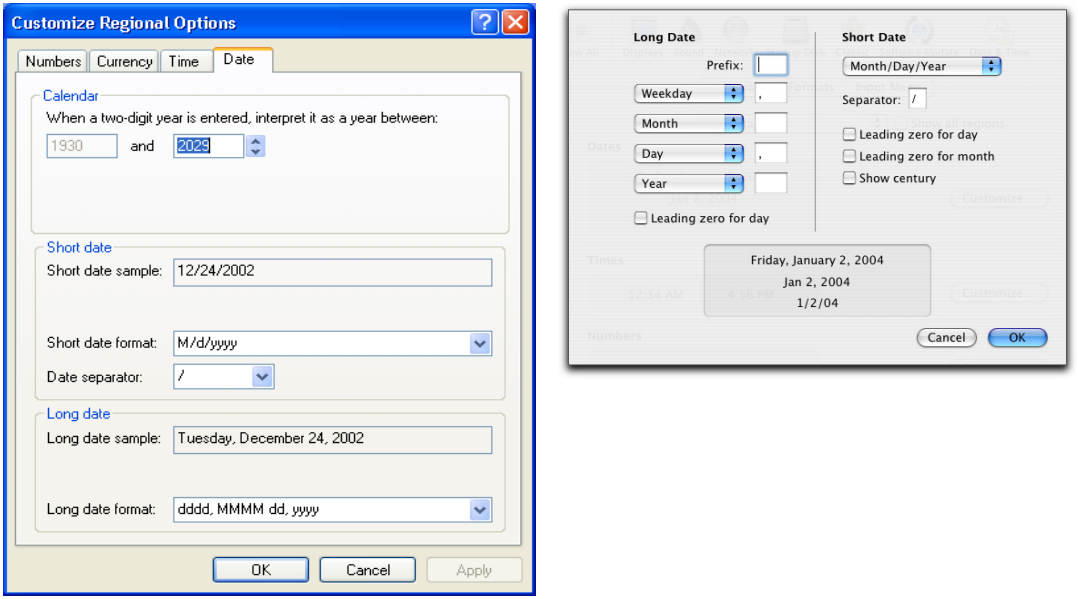
Table 22: Formatting properties of Date objects.

Property	Example
ShortDate	12/31/97
LongDate	Wednesday, December 31, 1997
AbbreviatedDate	Wed, Dec 31, 1997

Several aspects of the Long and Short date formats are controlled by the user's Date Properties (Windows) or Date Formats (Macintosh) settings. The Date Formats dialog is accessed from the Formats panel of the International system preference. On Windows, Date Properties is a screen in the Regional Settings control panel. Users can choose the order of the day, month, year, as well as the separators.

These screens are shown in Figure 267 on page 339.

Figure 267. The Date Formats and Date Properties screens.



To get the current date in any of these formats, simply create and instantiate a date object and then access the appropriate property. In this example, the current date formatted as a long date, is assigned to a variable:

```
Dim today as New Date
Dim theDate as String
theDate=today.LongDate
```

The TotalSeconds property of a Date object is the 'master' property that stores the date/time associated with the object. The TotalSeconds property is defined as the number of seconds since 1/1/1904.

Other property values are derived from TotalSeconds. If you change the value of the TotalSeconds property, the values of the Year, Month, Day, Hour, Minute, and Second properties change to reflect the second on which TotalSeconds occurs. Conversely, if you change any of these properties, the value of TotalSeconds changes commensurately.

In Windows applications, you must take care not to set the Day property to a day value that is greater than the days in the current month. When you want to assign such a value, you should first set the Day property to a legal value for any month and then set the remaining properties, resetting the Day property last (if needed). Another way to deal with the problem is to work directly with the TotalSeconds property and then access the properties that are derived from TotalSeconds.

Times

Time values are stored as part of a date. Date objects have two properties that store time values in two different formats. Table 23 lists the two properties and shows examples of how the time is returned.

Table 23: Formatting properties of Time objects.

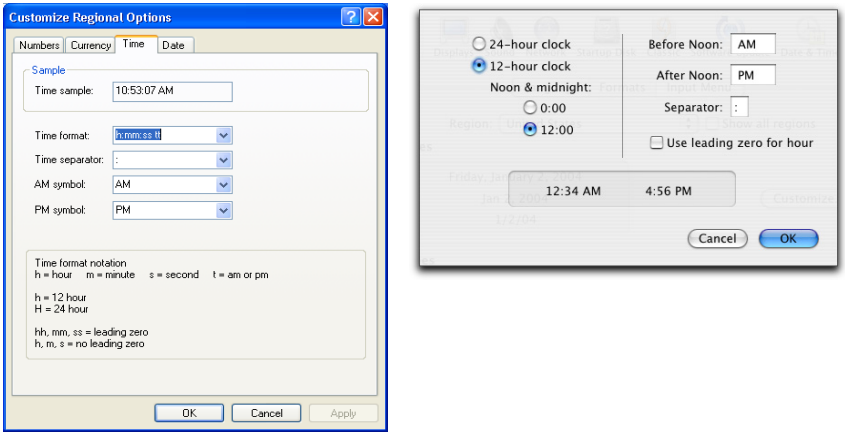
Property	Example
ShortTime	2:32 PM
LongTime	2:32:34 PM

To get the current time in either of these formats, create and instantiate a date object and then access the appropriate property. In this example, the current time formatted as a LongTime, is assigned to a variable:

```
Dim today as New Date
Dim Now as String
Now=today.LongTime
```

As is the case with date formats, several aspects of the Short Time and Long Time formats are controlled by the user via the Time screen in the Regional Settings Control panel on Windows or the Time Formats panel in International preferences (Macintosh). These screens are shown in Figure 268 on page 341.

Figure 268. The Time Formats and Time Properties screens.



Searching using Regular Expressions

REALbasic enables you to search and replace text using *regular expressions*. Regular expressions use a meta-language in which you can search for special characters, specific characters (e.g., only vowels), and search by position (e.g., at the beginning or end of a line). You use the language to define the string to search for and (optionally) the replacement string.

You use the properties of the RegEx class to define a regular expression search/replace or search operation. Table 24 shows these properties:

Table 24: Properties of the RegEx class.

Name	Description
Options	These options are various states which you can set for the Regular Expressions engine. See Table 26.
ReplacementPattern	This is the replacement string, which can include references to substrings matched previously, via the standard '\1' or '$1' notation common in regular expressions. This pattern is used either with the Replace property or passed to the RegExMatch class when Search returns, and subsequently used with Replace if no parameters are specified.
SearchPattern	This is the pattern you are currently searching for.
SearchStartPosition	Character position at which you want to start the search if the optional TargetString parameter to *Replace* is not specified. Keep in mind if you set it, it will only be used if you don't specify a TargetString, since setting a newtargetString resets the value.

The methods of the RegEx class, shown in Table 25, are used to do the find and replace.

Table 25: Methods of the RegEx class.

Name	Parameters	Description
Replace	Optional: TargetString as String; SearchStartPosition as Integer	Finds *SearchPattern* in Target, Replaces the contents of *SearchPattern* with *ReplacementPattern*, Returns the resulting String. Replace can take the optional parameters shown at left. Returns a String.
Search	TargetString as String SearchStartPosition as Integer	Finds *SearchPattern* in *TargetString*, if it succeeds it returns a RegExMatch. The RegExMatch will remember the ReplacementPattern specified at the time of the search. Returns a RegExMatch.

The RegExOptions class lets you specify the options shown in Table 26:

Table 26: Regular Expression options.

Name	Description
CaseSensitive	Specifies if case is to be considered when matching a string. Default is False.
DotMatchAll	Normally the period matches everything except a new line, this option allows it to match new lines.
Greedy	Greedy means the search finds everything from the beginning of the first delimiter to the end of the last delimiter and everything in-between. For example, Say you want to match the following bold-tagged text in HTML: The quick brown fox jumped If you use this pattern: .+ You end up matching "quick brown fox", which isn't what you wanted. So, you can turn *Greedy* off or use this syntax: .+? and you will match "quick", which is exactly what you wanted.)
LineEndType	This is in effect for the current Regular Expression "session") Has no effect on SearchPatterns if TreatAsOneLine is True. Changes the way \n is expanded for ReplacementPatterns 0 = any line ending (Mac or Win32 or Unix) 1 = defaultForPlatform (if running on Mac same as 2) (if running on Win32 same as 3) 2 = Mac ASCII 13 or \r 3 = Win32 ASCII 10 or \n 4 = Unix ASCII 10 or \n

Table 26: Regular Expression options.

Name	Description
MatchEmpty	Indicates whether patterns are allowed to match the empty string.
ReplaceAllMatches	Indicates whether all occurrences of the pattern are to be replaced.
StringBeginIsLineBegin	Indicates whether a string's beginning should be counted as the beginning of a line.
StringEndIsLineEnd	Indicates whether a string's end should be counted as the end of a line.
TreatTargetAsOneLIne	Ignores internal newlines for purposes of matching against '^' and '$'.
UTF8	If True, treats all processed characters as UTF8 characters. If False, ASCII is assumed.

When you find a match using a regular expression search, you use the properties of the RegExMatch class to obtain the matching string (or strings) and optionally replace the match with a string that you provide. The properties of the RegExMatch class are shown below:

Table 27: Properties of the RegExMatch class.

Name	Description
SubExpressionCount	Number of SubExpressions that are available with the search just performed. SubExpressions allow replacement of parts of the pattern.
SubExpressionString	Returns the SubExpression as a string for the given matchNumber. 0 returns the entire MatchString (the implicit 0^{th} subExpression), and 1 is the first real subExpression.
SubExpressionStart	Returns the starting position of the subExpression given by the matchNumber parameter.
Replace	Substitutes the matched result in a manner specified by the given *ReplacementPattern*. If no *ReplacementPattern* is specified, it uses the ReplacementPattern which was specified in the RegEx object at the time of the search.

See the entries in the *Language Reference* for more information about these three classes and a description of the syntax of regular expressions.

Adding Pictures and Drawing Graphics

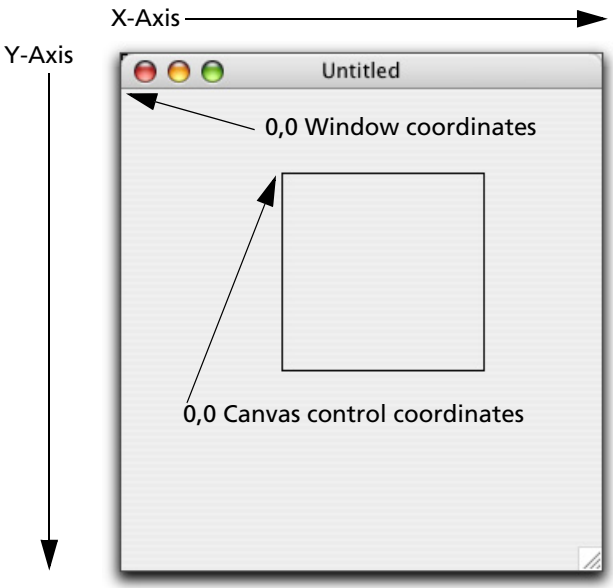
You can add pictures from documents or draw your own pictures in REALbasic. In some cases you can add the graphics you want without writing any code. When you do need to write code, REALbasic provides methods for creating all kinds of graphics.

Understanding the Coordinates System

Most of the graphics methods require you to indicate the location inside the window or within a Canvas control where you wish to begin drawing. This location is specified using the coordinates system. This system is a grid of invisible horizontal and vertical lines that are 1 pixel apart. If you have never done a computer drawing with a coordinates system, you might expect the origin (0,0) to be in the center of the window, but it's not. The origin is always in the upper-left corner of the area. For the entire screen, this is the upper-left corner of the screen. For a window, the origin is the upper-left corner of the window, and for a control, it's the upper-left corner of the control. The X axis (the horizontal axis) increases in value moving from left to right and the Y axis (the vertical axis) increases in value moving from top to bottom.

So, a point that is at 10, 20 (within a window) is 10 pixels from the left side of the window and 20 pixels from the top of the window. If you are working within a Canvas control, the point 10, 20 is 10 pixels from the left edge of the control and 20 pixels down from the top edge of the control.

Figure 269. The X,Y Coordinates System.



Displaying Pictures In a Window

There are different techniques you use to display pictures in a window. The technique you use depends on what you plan to do with the picture.

Using the Entire Window

If you want to use a window to display a picture, the window's Backdrop property is one way to do it. The Backdrop property is a picture that will be displayed behind

any controls in the window. By default, the Backdrop is set to "None" meaning that no Backdrop picture will be displayed.

There are several ways to assign a picture to a window's BackDrop property. On Macintosh, the most direct and intuitive way is to simply drag a picture document from the desktop to the window in the REALbasic IDE. When you do so, several things happen: The picture appears immediately in the window in the Window Editor, the name of the picture document is added to the Project Editor, and it is assigned to the BackDrop property in the window's Properties pane.

You can also assign a picture to a window's Backdrop property by dragging a picture document into your Project Editor and then choosing it by name as the picture for the Backdrop property in the Properties pane. For example, in a window's Open event handler, you can write:

```
Backdrop=picturename
```

where *picturename* is the name of the image, as it appears in the Project Editor.

In addition, you can set the Backdrop property at runtime by loading a picture via code or creating a new picture using the Graphics class drawing methods. This example presents the standard open file dialog box and lets the user choose a PICT, JPEG, or GIF file to be used as the backdrop of the current window[1]:

```
Dim f as FolderItem
f=GetOpenFolderItem("image/gif;image/jpeg;image/x-pict")
If f<> Nil Then
  Backdrop=f.OpenAsPicture
End If
```

After you have assigned a picture to the BackDrop property, you can then resize the window to the size of the picture by setting the window's width and height properties to the backdrop's width and height properties:

```
width=Backdrop.width
height=Backdrop.height
```

You don't need to worry about redrawing the Backdrop. REALbasic will handle redrawing the Backdrop when necessary.

Using a Portion of the Window

If you only want to display the picture in an area in the window, you can use an ImageWell control. An ImageWell is similar to a Canvas control, except that it has no drawing tools: you can only display an image that has been created elsewhere.

1. The file types used as parameters in the GetFolderItem call must be defined in the File Types dialog box.

To assign a picture to an ImageWell's Image property, simply drag it from the desktop to the Project Editor and then assign it to the ImageWell's Image property using its Properties pane.

You can also add a picture at runtime by loading an image using code. For example, the following code displays an open-file dialog box that allows the user to choose a PICT, JPEG, or GIF file and display it in the ImageWell. It is assumed that the file types used as parameters in GetOpenFolderItem have been assigned in the File Type Sets Editor or with the FileType class via the language.

```
dim f as FolderItem
f=GetOpenFolderItem("image/x-pict;image/jpeg;image/gif")
if f <> Nil then
 ImageWell1.Image=f.OpenAsPicture
end if
```

You can also allow your users to drag a picture document from the Finder to the ImageWell rather than using the open-file dialog box. In the ImageWell's Open event handler, allow a file drop using the lines:

```
me.acceptfileDrop("image/x-pict")
me.acceptfileDrop("image/jpeg")
me.acceptfileDrop("image/gif")
```

In the ImageWell's DropObject event handler, use the code:

```
Sub (DropObject (Obj as DragItem)
 If Obj.FolderItemAvailable then
  Me.Image=Obj.FolderItem.OpenAsPicture
 End if
```

NOTE: The code will only support PICT files on Macintosh and .bmp files on Windows unless QuickTime is installed on the user's computer.

Your other option is to use a Canvas control to display a picture in a portion of the window. This type of control also gives you a graphics area that can be drawn in and also receives events. You might use a Canvas control if you need to display a picture that the user will interact with. With the Canvas control's Backdrop property you can display an existing picture. This can be done manually in the Window Editor by clicking on the Canvas control in a window to select it and then choosing a picture from the Backdrop property's pop-up menu in the Properties pane. The picture must have been added to the project in the Project Editor. A picture can also be assigned to the Backdrop property at runtime. This example displays an open-file dialog box

when the user clicks on the Canvas control and then lets the user choose a picture to be displayed in the Canvas control:

```
Dim f as FolderItem
f=GetOpenFolderItem("image/x-pict")
If f<> Nil Then
 Me.Backdrop=f.OpenAsPicture
End If
```

You can also support drag and drop to a Canvas control in the same manner as for ImageWells. The advantage of using a Canvas control is that you can also customize the image using the methods of the Graphics class using the Canvas control's Paint event handler. This feature is described in the following section.

Creating Pictures

You can create pictures programmatically using the methods of the Graphics class. A Graphics object is simply an object in memory that holds an image. For example, windows and Canvas controls have a Paint event. This event is executed any time the window or Canvas control needs to be redrawn. For example, when a window opens, its Paint event is executed because the contents of the window needs to be drawn. Any Canvas controls in a window will also execute their Paint event when the window opens because the Canvas control needs to be drawn. These Paint events are also executed when a portion of the window and/or Canvas control that was previously hidden by another window is exposed.

The Paint event is passed a Graphics object. When the Paint event is finished executing, this graphics object will be drawn in the window or Canvas control. You draw in a window or Canvas control by calling the drawing methods of this graphics object.

Displaying Pictures

You can display a picture in a Graphics object using the DrawPicture method of the Graphics class. This method is passed a picture and the coordinates that describe where you want the picture drawn within the graphics object. This example uses the Paint event to draw two pictures that have been dragged into the Project Editor (BartPict and LisaPict) side by side:

```
Sub Paint(g As Graphics)
  g.DrawPicture BartPict, 0,0
  g.DrawPicture LisaPict,BartPict.Width, 0
```

Copying A Portion of a Picture

The DrawPicture method of the Graphics class can be used to copy a portion of a picture to a Graphics object. This is done using the optional parameters of the DrawPicture method. These parameters allow you to specify the portion of the picture you want to draw. You can specify the coordinates where you wish to begin copying from the picture as well as the amount (in width and height) you wish to copy.

This example draws a 20 pixel square portion of the source picture starting 10 pixels from the left and 10 pixels from the top of the source picture and drawing the picture 5 pixels from the left and 5 pixels from the top of the Canvas control or window background:

```
Sub Paint(g As Graphics)
  g.DrawPicture Lisa,5,5,Lisa.width,Lisa.height,10,10,20,20
```

Scaling Pictures

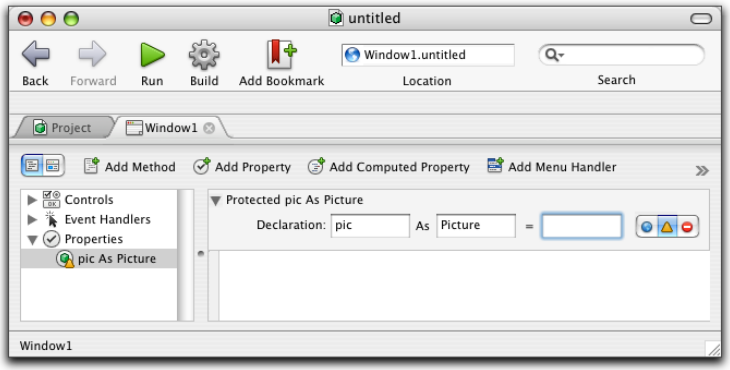
The DrawPicture method of the Graphics class can scale a picture when it is drawn. To do this, you must include all of the DrawPicture parameters. Scaling is done by specifying a destination width and/or height that is larger or smaller than the picture's original width and/or height. This example draws a picture at two times its original size:

```
Sub Paint(g As Graphics)
  Dim w,l as integer
  w=lisa.width
  l=lisa.height
  g.DrawPicture lisa, 0,0,w*2,l*2,0,0, lisa.width,lisa.height
```

Here's a very useful example that automatically scales an imported picture to fit in a Canvas control.

Assume that you have a window that has a Canvas control and a PushButton (or other control) that triggers code that does the import and scaling.

Create a property of the parent window that will hold the scaled picture. For example:



In the Action event of the PushButton, use the following code to import the picture and do the scaling:

```
Dim f As FolderItem
Dim p As Picture
Dim maxWidth, maxHeight As Integer
Dim factor As Double

maxWidth = Canvas1.width
maxHeight = Canvas1.height

f = GetOpenFolderItem("PICT")
If f <> Nil then
  p = f.OpenAsPicture
end if

factor = Min( maxWidth / p.Width, maxHeight / p.Height )
factor = Min( factor, 1.0 )  // (don't scale it up if it's too small!)

pic = NewPicture( p.Width * factor, p.Height * factor, 32 )
pic.graphics.DrawPicture p, 0,0,pic.width,pic.height, 0,0,p.width,p.height

Canvas1.Refresh
```

(This assumes you want to import a PICT file and have defined a file type of PICT in the File Type Sets Editor or with the FileType class via the language.)

The values of maxWidth and maxHeight are set in this example to the size of the Canvas control, but you can, of course, supply other values. The DrawPicture method of the Graphics class uses the following parameters:

Parameter	Type	Description
Image	Picture	The picture to be drawn at the location specified by X and Y.
X	Integer	X coordinate within the control of the left side of the image. Defaults to zero.
Y	Integer	Y coordinate within the control of the top of the image. Defaults to zero.
DestWidth	Integer	Width of Image within the control
DestHeight	Integer	Height of Image within the control
SourceX	Integer	X coordinate of the picture you copy from. Defaults to zero.
SourceY	Integer	Y coordinate of the picture you copy from. Defaults to zero.
SourceWidth	Integer	Width of the picture you wish to copy. Defaults to width of the picture.
SourceHeight	Integer	Height of the picture you wish to copy. Defaults to height of the picture.

The final step is to add code to the Paint event of the Canvas control to actually assign the scaled image, pic, to the Backdrop property of the Canvas.

```
If pic <>Nil then
  g.DrawPicture pic,0,0
End if
```

Drawing Standard Dialog Icons

REALbasic has a MsgBox function for displaying a standard message box with a note icon and an OK button. However, there may be times when this isn't appropriate. For example, the note icon is appropriate when you need to inform the user about something that isn't a warning. If the user is about to do something where data loss could occur (like quitting the application without saving a changed document), then the caution icon is more appropriate. If the user has started an operation that cannot be completed (such as saving a document to a locked volume), the stop icon is more appropriate.

Figure 270. Mac OS X, Windows, and Linux Note, Caution, and Stop icons.

NOTE: Under Mac OS classic, the look of these three icons may be affected by the end-user's choice of Appearance Manager schemes or Kaleidoscope themes. They also may look different under Windows skins and different Linux desktops.

The Graphics class provides the DrawNoteIcon, DrawCautionIcon, and DrawStopIcon methods that make it easy to display these icons in a Canvas control or a window background. These methods make system calls that draw the correct icon for the current version of the operating system. Using these methods, you will display the appropriate icon for the user's platform. The following example draws the note icon in a Canvas control in its Paint event:

```
Sub Paint(g As Graphics)
  g.DrawNoteIcon 0,0
```

If you need to use these icons in a message dialog box, you can get them via the MessageDialog class. The MessageDialog class can create a dialog box with a main text message, a subordinate explanation, an icon, and one to three buttons. The user must respond to the dialog by clicking one of the buttons and your code can detect the button the user clicked and take appropriate action.

You can set the icon to be displayed simply by setting the value of the MessageDialog's Icon property. The MessageDialog object positions the icon, text, and buttons for you. For more information about the MessageDialog class, see the section "Message Dialog Boxes" on page 79 and the entry for the MessageDialog class in the *Language Reference*.

Drawing Pixels

You can get and set the color of individual pixels in a Graphics object using the Pixel property. You use this property by passing it X and Y coordinates and then setting the color of that pixel to a color object or getting its color.

This example draws pixels at randomly selected coordindates within a Graphics object using randomly selected colors until the user presses Escape or ⌘-Period (Macintosh):

```
Sub Paint(g As Graphics)
  Dim c as Color
  Do
    c=Rgb(Rnd*255,Rnd*255,Rnd*255)
    g.Pixel(Rnd*me.Width,Rnd*me.Height)=c
  Loop until UserCancelled
```

This example gets the color of the pixel the mouse is over in a Canvas control and fills another Canvas control called PixelColor with that color:

```
Sub MouseMove(X As Integer, Y As Integer)
  Dim c as Color
  c=Me.Graphics.Pixel(X,Y)
  PixelColor.Graphics.ForeColor=c
  PixelColor.Graphics.FillRect 0,0,PixelColor.Width,PixelColor.Height
```

Drawing Lines

Lines are drawn using the DrawLine method of the Graphics class. The color of the line is the color stored in the ForeColor property of the Graphics object the line is being drawn in. To use the DrawLine method, you pass it starting coordinates and ending coordinates of the line.

This example uses the DrawLine method to draw a grid inside a Canvas control or window background. The size of each box in the grid is defined by the value of the boxSize variable:

```
Sub Paint(g as Graphics)
  Dim i, boxSize as Integer
  boxSize=10
  For i=boxSize to Me.Width Step boxSize
    g.DrawLine i,0,i,Me.Height
  Next
  For i=boxSize to Me.Height Step boxSize
    g.DrawLine 0,i,Me.Width,i
  Next
```

The thickness of the line is controlled by the PenHeight and PenWidth properties of the Graphics object.

Drawing Ovals

Ovals are drawn with the DrawOval and FillOval methods of the Graphics class. Both require the same parameters: the X and Y coordinates where the oval starts and the width and height of the oval. Both draw ovals using the ForeColor property of the Graphics object. Both use the PenWidth and PenHeight properties of the Graphics object to determine the line thickness. The difference between the two is that DrawOval draws only the border of the oval, leaving the interior blank. FillOval draws an oval with the interior filled with the ForeColor.

This example draws an oval in a Canvas control or Window background:

```
Sub Paint(g as Graphics)
  g.DrawOval 0,0,50,75
```

Drawing Rectangles

Rectangles are drawn using the DrawRect, FillRect, DrawRoundRect, and FillRoundRect methods of the Graphics class. All of these methods use the ForeColor property of the Graphics object and the PenWidth and PenHeight properties to determine the line thickness. All of these methods require the X and Y coordinates of the upper-left corner of the rectangle, as well as the width and height of the rectangle. RoundRectangles are rectangles with rounded corners. Therefore, DrawRoundRect and FillRoundRect require two additional parameters: the width and height of the curve of the corners.

DrawRect and DrawRoundRect both draw empty rectangles. FillRect and FillRoundRect draw solid rectangles.

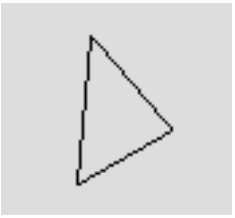
Drawing Polygons

Polygons are drawn using the DrawPolygon and FillPolygon methods of the Graphics class. Polygons are drawn by passing the DrawPolygon or FillPolygon method an integer array that contains each point in the polygon. This is a 1-based array where odd numbered array elements contain X values and even numbered array elements contain Y coordinates. This means that element 1 contains the X coordinate of the

first point in the polygon and element 2 contains the Y coordinate of the first point in the polygon. Consider the following array values:

Element #	Value
1	10
2	5
3	40
4	40
5	5
6	60

When passed to the DrawPolygon or FillPolygon method, this array would draw a polygon by drawing a line starting at 10,5 and ending at 40,40 then drawing another line starting from 40,40 ending at 5,60 and finally a line from 5,60 back to 10,5 to complete the polygon. This polygon has only three sets of coordinates so it is a triangle.



The code in the Paint event of a Canvas control or Window to draw this polygon, looks like this:

```
Sub Paint(g As Graphics)
  Dim points(6) as Integer
  points(1)=10
  points(2)=5
  points(3)=40
  points(4)=40
  points(5)=5
  points(6)=60
  g.DrawPolygon points
```

FillPolygon draws the same polygon but with the interior filled with the ForeColor:

Another way of populating the array is with the Array function. The Array function takes a list of values separated by commas and populates the array, beginning with element zero. Since the first element that DrawPolygon uses is element 1, you can use any value in element zero:

```
Sub Paint(g As Graphics)
  Dim points() as Integer
  points=Array(0,10,5,40,40,5,60)
  g.DrawPolygon points
```

Creating Custom Controls with the Canvas Control

Visible controls (controls that have a graphical interface the user can interact with directly, like PushButtons) are pictures that have code that controls how they are drawn. This means that a Canvas control can easily be used to create controls that are not built-in to REALbasic.

Suppose you wanted to create a simple custom control like a rectangle whose fill color toggles from black to white when clicked. First you would drag a Canvas control into a window. You want the rectangle to switch colors when the user clicks the mouse, so this code goes in the MouseDown event handler of the Canvas control. The code checks to see if the rectangle is black and, if it is, fill it in white; otherwise fill it in black. You can check the color of any particular pixel using the Pixel property of the graphics property of the Canvas control. You can determine if a pixel is a particular color by comparing it to a color value returned by the Rgb function. Passing 0 (zero) to each of the parameters of the Rgb function returns the color black. Passing 255 to each parameter of the Rgb function returns the color white. You will learn more about color later in this chapter. So, the code for the MouseDown event handler looks like this:

```
Function MouseDown(X As Integer, Y As Integer) As Boolean
  If Me.Graphics.Pixel(X,Y)=Rgb(0,0,0) Then
    Me.Graphics.ForeColor=Rgb(255,255,255)
  Else
    Me.Graphics.ForeColor=Rgb(0,0,0) //black
  End If
  Me.Graphics.FillRect Me.Left,Me.Top,Me.Width,Me.Height
```

This code checks to see if the pixel the user clicked on is black and, if it is, the ForeColor property of the graphics object of the Canvas control (generically represented here using the Me function) is set to white, otherwise it's set to black. Next, the FillRect method of the Graphics property of the Canvas control is called to fill the rectangle with the color stored in the ForeColor property.

There's one more step before our custom control is complete. If the Canvas control needs to be redrawn for some reason (such as when the window first opens or the user moves another window in front of the one with the Canvas control), REALbasic calls the Canvas control's Paint event handler to redraw the Canvas control. If there is no code in the Paint event handler, REALbasic won't draw the rectangle and, to

the user it will seem to appear and disappear at different times, which will be confusing. To solve this problem, you need to put a slightly altered version of the code you have in the MouseDown event handler in the Paint event handler:

```
Sub Paint(g As Graphics)
  If g.Pixel(0,0)=Rgb(0,0,0) Then
    g.ForeColor=Rgb(255,255,255)
  Else
    g.ForeColor=Rgb(0,0,0)
  End If
  g.FillRect Me.Left,Me.Top,Me.Width,Me.Height
```

Since the Paint event handler is passed a reference to the Graphics object of the Canvas (the g parameter), you can make the code a bit more generic and use "g" instead of "me.graphics". Also, since the user isn't clicking anywhere, you need to choose a pixel whose color you check. In this example we chose the pixel at 0,0.

This is an example of a very simple custom control. More complex and generic controls can be created using classes. See "Creating Custom Interface Controls with Classes" on page 455 for more information.

Working with Vector Graphics

The REALbasic language includes a group of classes that enable you to create, open, and save vector graphics. A vector graphic (as opposed to a bitmap graphic) is composed entirely of primitive objects—lines, rectangles, text, circles and ovals, and so forth—that retain their identity in the graphic. They don't "decompose" and become part of an indistinguishable bitmap. The Object2D class in the REALbasic language is the base class for all the classes that create primitive objects. They are shown in Table 28:

Table 28: Classes used to draw vector graphics.

Class	Description
ArcShape	Draws an arc of a circle.
CurveShape	Draws straight lines or curves using one or more "control points."
FigureShape	Draws polygons that can (optionally) have curved sides.
OvalShape	Draws circles and ovals.
PixMapShape	Imports a bitmap picture into the image.
RectShape	Draws a square or rectangle.
RoundRectShape	Draws a square or rectangle with rounded corners (subclassed from RectShape).
StringShape	Draws text strings in a specified font, font size, and style.

Since each class in Table 28 is subclassed from Object2D, you can use the Object2D's properties to draw the object. They are shown in Table 29.

Table 29: Properties of the Object2D class.

Name	Description
Border	Opacity of the border, from 0 (transparent) to 100 (opaque). The default is 0.
BorderColor	Color of the border.
BorderWidth	Width of the border, in points. The default is 1.
Fill	Opacity of the interior, from 0 (transparent) to 100 (opaque). The default is 100.
FillColor	Color of the interior of the shape.
Rotation	Clockwise rotation, in radians, around the X, Y point.
Scale	Scaling factor relative to the object's original size.
X	Horizontal position of center or main anchor point.
Y	Vertical position (down from top) position of center or anchor point.

Each class in Table 28, of course, has additional properties that pertain to the object's particular shape or characteristics. For example, the RectShape class has additional properties for the rectangle's height and width. The RoundRectShape class adds properties for specifying the width and height of the rounded corners and the number of straight line segments used to approximate the curved corners. Please refer to the entries in the *Language Reference* for information on each property.

Drawing and Displaying a Vector Object

You draw a single vector object simply by instantiating it and specifying its properties. For example, the following code draws a RoundRectShape:

```
Dim r as New RoundRectShape
r.width=120
r.height=120
r.border=100
r.bordercolor=RGB(0,0,0) //black border
r.fillcolor=RGB(255,102,102)
r.cornerHeight=15
r.cornerWidth=15
r.borderwidth=2.5
```

The only problem with this is that the shape doesn't appear anywhere. It's just "defined"— ready for your use. You need to add a command to draw the vector object in another object such as a window or a control that can display graphics such as a Canvas control.

The Paint event of a control or window is a good place to insert code that draws vector objects. The Paint event passes a Graphics object to the event handler, so you

only need to call the DrawObject method of the Graphics class to draw the object. Access to the Graphics class is provided by the passed parameter, g. The DrawObject method takes three parameters, the object to be drawn and its X and Y coordinates in the Graphics space.

The following code in the Paint event of a Window draws the finished vector object:

```
Dim r as New RoundRectShape
r.width=120
r.height=120
r.border=100
r.bordercolor=RGB(0,0,0) //black border
r.fillcolor=RGB(255,102,102)
r.cornerHeight=15
r.cornerWidth=15
r.borderwidth=2.5
g.DrawObject r,100,100 //draw at 100,100
```

You can also create composite vector graphics objects that are made up of several individual vector graphics objects. The composite object is a Group2D object—it's just a group of Object2D objects. Use the Append or Insert methods of the Group2D class to add individual vector graphic objects to the Group2D object. When you are finished, draw the object using one call to the DrawObject method.

The following code illustrates this. We've taken the code shown above and added a second RoundRectShape object and moved it 20 pixels to the right and 20 pixels down from the original RoundRectShape. The two Append statements create the

composite object. The code is in the Paint event of a window, so the parameter, g, provides access to the Graphics class.

```
Dim r as New RoundRectShape
Dim s as New RoundRectShape
Dim group as New Group2D

r.width=120
r.height=120
r.border=100
r.bordercolor=RGB(0,0,0) //black border
r.fillcolor=RGB(255,102,102)
r.cornerHeight=15
r.cornerWidth=15
r.borderwidth=2.5

s.width=120
s.height=120
s.border=100
s.bordercolor=RGB(0,0,0) //black border
s.fillcolor=RGB(255,102,102)
s.cornerHeight=15
s.cornerWidth=15
s.borderWidth=2.5

s.x=r.x+20   //shift s 20 pixels to right
s.y=r.y+20  //shift s 20 pixels down

group.append r
group.append s
g.drawObject group,100,100
```

When you want to add vector graphics to an existing bitmap image, you create a Group2D object and then add each object to the Group2D using its Append

method. This example appends a StringShape to a PixmapShape. The graphic, h1, has been dragged to the Project Editor.

```
Dim px as PixmapShape
Dim s as StringShape
Dim d as New Group2D

px=New PixmapShape(h1) //h1 is a graphic in the Project Editor
d.append px

s=New StringShape
s.y=70
s.Text="This is what I call a REAL car!"
s.TextFont="Helvetica"
s.Bold=true
d.append s
graphics.drawobject d,100,100
```

Opening and Saving Vector Graphics

Two methods of the FolderItem class are relevant to vector graphics—OpenAsVectorPicture and SaveAsPicture. The OpenAsVectorPicture method opens a PICT file (Macintosh) or an .emf file (Windows) and attempts to convert the objects in the PICT to editable REALbasic Object2D objects. The original file may contain certain elements that do not have a REALbasic equivalent, but OpenAsVectorPicture will do its best to map these objects.

The SaveAsPicture has an optional parameter that specifies the format to use when doing the save. In particular, you can pass a code that instructs REALbasic to save in the PICT format (Macintosh) or the .emf or .bmp formats (Windows) while preserving the vector graphic information. Please refer to the FolderItem entry in the Language Reference for information on the values of the optional parameter.

Working With Color

Color in REALbasic is a data type. It consists of three values that define a color. A color can be specified using either the RGB, HSV, or CMY models. You use the three relevant Color properties to set the color. For example, to use the RGB model, use the RGB function and set values for the Red, Green, and Blue properties. These values range from 0 to 255. The RGB function returns a Color when passed values for the amount of red, green, and blue. Several classes have Color properties. For example, the ForeColor property of the Graphics class is a Color.

If you need to store a Color, you can create a property or variable of type Color and then use the RGB, HSV, or CMY function. In this example, a new variable of type Color is created and the values for the white are assigned using the RGB function:

```
Dim c as Color
c=Rgb(255,255,255)
```

You can also assign a color value directly, without using the RGB, HSV, or CMY functions. You use the RGB color model with the following syntax to specify a color:

```
&cRRGGBB
```

where RR is the hexadecimal value for Red, GG is the hexadecimal value for Green, and BB is the hexadecimal value for Blue. Each value goes from 00 to FF rather than 0 to 255. For example, the following is equivalent to the previous example:

```
Dim c as Color
c=&cFFFFFF
```

"FF" is hexadecimal for 255. There is an easy way to obtain the hexadecimal values. Using the Add Constant declaration area, you can define a constant of type color and use the built-in Color Picker to choose a color visually. When you select a color, REALbasic figures out the hex values and inserts them in the Value area of the dialog box. For an example, see the section "Adding a Constant to a Module" on page 305.

In this example, the ForeColor property of a Graphics object is set to blue so the text drawn will be in that color:

```
Sub Paint(g as Graphics)
  g.ForeColor=RGB(9,13,80)
  g.DrawString "Hello World",50,50
```

Determining The RGB Values For a Color

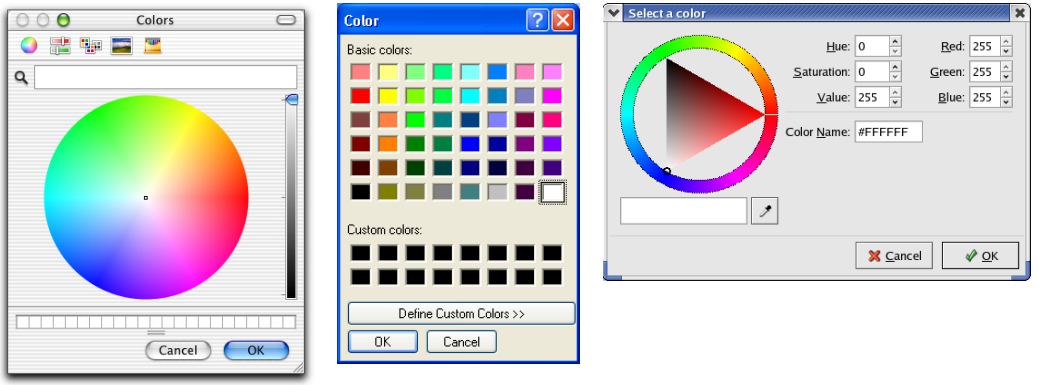
If you need to assign a color at runtime but aren't sure which RGB values to use to get a particular color, you can use the Mac OS Color Picker. The following code displays the Color Picker:

```
Dim c as Color
Dim b as Boolean
b=SelectColor(c,"Choose a color")
if b then //user chose a color
// do something with selected color here
end if
```

If the user cancelled out of the Color Picker dialog box, the boolean variable, b, is False; otherwise, the selected color is returned in the color object, c, and is available for assignment to a color property of an object.

You may have already used the Color Picker to assign a color to a control's property. If you haven't, the Color Picker displays a color palette and allows you to click on one to pick it (hence the name). Figure 271 on page 361 shows the Color Pickers on Mac OS X, Windows, and Linux.

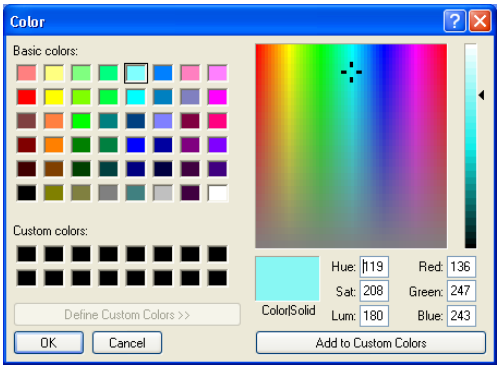
Figure 271. The Macintosh, Windows, and Linux Color Pickers.



In the Mac OS X color pickers, you click on a color and a sample of the color appears in the patch area at the top of the screen. When you click OK, REALbasic translates your selection into RGB values.

The Windows version of the Color Picker uses only one format, shown in Figure 271. You can either select one of the predefined colors or click the Define Custom Colors button to display the "advanced" color picker, which depicts colors on a continuum and lets you specify the color using either the RGB or HSV models.

Figure 272. The Windows Custom Colors Picker.



Select a color by clicking on a point in the color spectrum or enter values in the RGB or HSV areas. Click Add to Custom Colors to add the custom color to one of the Custom Color samples on the left side of this dialog.

The Pixel Property of Graphics Objects

The Pixel property of a Graphics object lets you get and set the color of the pixel you specify. This property is an example of a property whose data type is Color. In

this example, the Paint event handler is setting a pixel to black if it is white and white if it is black:

```
Sub Paint(g As Graphics)
  If g.Pixel(10,20)=Rgb(0,0,0) Then
    g.Pixel(10,20)=Rgb(255,255,255)
  Else
    g.Pixel(10,20)=Rgb(0,0,0)
  End if
```

You can see that the code to check the color of a pixel and set the color of a pixel is basically the same.

Printing Text and Graphics

REALbasic provides a lot of flexibility when it comes to printing. You can display the Page Setup dialog box and store the settings the user chooses.

Printing is almost exactly the same as drawing text and graphics into a Canvas control or the graphics property of a Window. When you call the OpenPrinter or OpenPrinterDialog function, a Graphics object is returned. To print, you simply draw your text and graphics into this Graphics object. To cause the page to print, you call the NextPage method of the Graphics object. This method forces the Graphics object to be printed, then clears it so you can use it again to draw the next page.

Working with the Page Setup Dialog Box

The PrinterSetup class lets you create an object that can be used to display the Page Setup dialog box, get and set the individual Page Setup settings, as well as store and restore these settings. To display the Page Setup dialog box, call the PageSetupDialog method of the PrinterSetup object you have instantiated. This method returns True if the user clicks the OK button in the Page Setup dialog box and False if he clicks the Cancel button. The PrinterSetup class has properties for accessing all of the settings in the Page Setup dialog box (page orientation, scale, etc.). For a list of PrinterSetup properties, the PrinterSetup class in the *Language Reference*. However, in most cases you won't have to deal with these properties because a composite version of these settings is stored in the SetupString property. The SetupString property is read/write and is used to get all of the PrinterSetup settings as a string so you can store them and to restore that string later on. For example, in a document-based application, a string property could be added to the document window that stores the SetupString value. When the user chooses to display the Page Setup dialog box (in most applications by choosing Page Setup from the File menu), a PrinterSetup object is created and its SetupString property is assigned the value in the window

property storing these settings. Then the Page Setup dialog box is displayed showing these settings. In this example, the window property is called "Settings":

```
Dim ps as PrinterSetup
ps=New PrinterSetup
ps.SetupString=Settings
If ps.PageSetupDialog Then
  Settings=ps.SetupString
End if
```

If the user clicks OK in the Page Setup dialog box, the window's Settings property is assigned the value of the SetupString because settings in the Page Setup dialog box may have been changed by the user.

PrinterSetup class objects can be optionally passed as a parameter to the OpenPrinter and OpenPrinterDialog functions so that the Page Setup settings can be used during printing.

If you wish to store the PrinterSetup's SetupString property with the document when the user saves the document (assuming you provide this capability), you will probably need to store it in a string resource in the resource fork of the document. See "Working With Macintosh Resources" on page 412 for more information on the resource fork.

The Page Setup dialog box is not supported in the initial release of REALbasic for Linux. Calling the PrinterSetup function will return False and no dialog will be presented to the user.

Printing With The Print Dialog Box

You use the OpenPrinterDialog function to display the Print dialog box and print. If the user clicks the OK button in the Print dialog box, a Graphics class object is returned. If the user clicks the Cancel button, the Graphics object returned will be Nil. To create the first page to be printed, you utilize the Graphics object returned, calling the various Graphics class methods such as DrawString, DrawLine, DrawOval, DrawPicture, etc. Once you have created the page, you can send the page to the printer by calling the NextPage method of the Graphics class. This method will both send the page to the printer for printing and clear the Graphics object so you can begin creating the next page.

This example displays the Print dialog box then prints "Hello" on the first page and "World" on the second page:

```
Dim page as Graphics
page=OpenPrinterDialog()
If page<> Nil Then
  page.DrawString "Hello", 50, 50
  page.NextPage
  page.DrawString "World", 50, 50
  page.NextPage
End
```

The Print dialog box page range is automatically supported. In the last example, if the user chooses to print pages 2 through 2, he will get only page 2.

If you are storing the SetupString property of the PrinterSetup class object, you can optionally pass this string to the OpenPrinterDialog function if you want it to use the settings stored in the SetupString. This example assumes that the SetupString is stored in a window property called "Settings" and passes it to the OpenPrinterDialog function for consideration during printing:

```
Dim page as Graphics
Dim ps as PrinterSetup
ps=New PrinterSetup
If Settings <> "" Then
  ps.SetupString=Settings
End If
page=OpenPrinterDialog(ps)
If page <> Nil Then
  page.DrawString "Hello", 50, 50
  page.NextPage
  page.DrawString "World", 50, 50
  page.NextPage
End If
```

For more information on the OpenPrinterDialog function, see the OpenPrinterSetup function in the *Language Reference*.

Printing Without The Print Dialog Box

To print without displaying the Print dialog box, call the OpenPrinter function. This function is identical to the OpenPrintDialog function except that it doesn't display the Print dialog box before printing. For information on printing, see the section "Printing With The Print Dialog Box" on page 363. For more information on the OpenPrinter function, see the OpenPrinter function in the *Language Reference*.

Printing Styled Text

Because EditFields are capable of displaying styled text and multiple font sizes, you will usually want to retain the styled text in your reports. The StyledTextPrinter class supports this capability. It uses the DrawBlock method (rather than the Draw-

String method) to accomplish this. Here is a simple example that prints the contents of an EditField as styled text (The StyledTextPrinter method returns Nil unless the EditField's MultiLine property is set to True. Of course, the user cannot enter styled text into an EditField unless the MultiLine and Styled properties are True.).

```
dim stp as styledTextPrinter
dim g as graphics
g=openPrinterDialog()
if g <> Nil then
  stp=EditField1.StyledTextPrinter(g,72*7.5)
  stp.drawBlock 0,0,72*9
end if
```

The parameters of DrawBlock are the top-left x, y coordinates on the page and the height of the block. This example starts at the top-left corner. See the description of the StyledTextPrinter class in the *Language Reference* for more information.

The REALbasic *Tutorial* includes a chapter on printing the styled text in an EditField. It includes code for printing more than one page and setting the printable area on the page. For more information, see Chapter 8, "Printing Styled Text" of the Tutorial.

Transferring Text and Graphics with the Clipboard

The Clipboard is a class of object in REALbasic with properties and methods. The properties and methods let you determine what kind of data is available on the Clipboard, get data from the Clipboard, and send data to the Clipboard. The Clipboard class supports three kinds of data: text, picture, and binary. Binary data is represented in string form and is marked with a type you specify so you can tell what the binary data represents.

NOTE: For EditFields, REALbasic handles the Cut, Copy, and Paste operations of the Edit menu automatically. However, for other controls that contain data such as Canvas and ListBox controls, this is not the case.

To access the Clipboard for any reason, you must first create a new object of type Clipboard:

```
Dim c as Clipboard
c=New Clipboard
```

In the event handler that opened the Clipboard, you must call the Clipboard object's Close method or an error may occur.

Testing The Clipboard For Specific Data Types

You can test the Clipboard using the following methods and properties all of which return True or False: TextAvailable, PictureAvailable, and RawDataAvailable. RawDataAvailable is used to determine if a specific kind of binary data (usually data put there by your application) is available. To use the RawDataAvailable method, you must pass it the MacType string that represents the type of data. This string was passed when the binary data was passed when the data was put on the Clipboard.

Getting Data From The Clipboard

Once you know what kind of data is available on the Clipboard, you can get the data using the Text, Picture, and RawData properties. In this example, if text is available, the text is placed in a variable called "Cliptext."

```
Dim c as Clipboard
Dim ClipText as String
c=New Clipboard
If c.TextAvailable Then
  ClipText=c.Text
End If
C.Close
```

If a picture is available, the picture is placed in a variable called "ClipPict."

```
Dim c as Clipboard
Dim ClipPict as Picture
c=New Clipboard
If c.PictureAvailable Then
  ClipPict=c.Picture
End If
C.Close
```

In this example, rows from a ListBox that have been copied to the Clipboard are added to a ListBox:

```
dim theRows as string
dim c as clipboard
c=New Clipboard
If c.RawDataAvailable("rows") Then
  theRows=c.RawData("rows")
  Do
    ListBox1.AddRow Left(theRows,InStr(theRows,EndOfLine)-1)
   theRows=Mid(theRows,InStr(theRows,EndOfLine)+1)
  Loop until theRows=""
End If
C.Close
```

NOTE: Remember, you must call the Clipboard object's Close method in the event handler that opened the Clipboard or an error may occur.

Putting Data On The Clipboard

You can put text, picture, or binary data (in the form of a string) on the Clipboard. To do this, you create a new Clipboard object then use the appropriate method or property based on the type of data you wish to put on the Clipboard.

Data Type	Method or Property
Text	SetText method
Picture	Picture property
Binary Data	AddRawData method

In this example, text is added to the Clipboard:

```
Dim c as Clipboard
c=New Clipboard
c.SetText "Hello World"
c.Close
```

In this example, a picture from Canvas1 is copied to the Clipboard:

```
Dim c as Clipboard
c=New Clipboard
c.Picture=Canvas1.Backdrop
c.Close
```

In this example, rows from a ListBox are copied to the Clipboard. They are copied using the AddRawData method so they don't appear as text on the Clipboard:

```
Dim i as Integer
Dim c as Clipboard
Dim rows as String
c=New Clipboard
For i=0 to ListCount
  If ListBox1.Selected(i) Then
    rows=rows+ListBox1.List(i)+EndofLine
  End If
Next
c.AddRawData rows,"rows"
c.Close
```

NOTE: Remember, you must call the Clipboard object's Close method in the event handler that opened the Clipboard or an error may occur.

Creating Animation with Sprites

The SpriteSurface control is used to create animation where pictures can be moved around the screen with all redrawing handled automatically by the SpriteSurface control. Each picture is a Sprite object. Sprite objects have x and y properties that determine their current location on the screen when the SpriteSurface is running the sprite animation.

Causing Sprites to Move and Change Images

The NextFrame event handler is called each time the SpriteSurface is ready to draw the next frame of animation. The NextFrame event handler is run each time the SpriteSurface's Update method is called.

If you want a Sprite to change position in the next frame, change its X and/or Y properties in the NextFrame event handler. If you want the Sprite's picture to change in the next frame of animation, change its Image property in the NextFrame event handler. To remove a Sprite from the animation, call the Sprite's Close method.

Frame Redrawing

The speed at which frames are redrawn is based on the FrameSpeed property. This property determines the number of times the monitor will refresh each second. This also determines the number of times per second that the NextFrame event handler will execute.

The FrameSpeed parameter is defined as the number of vertical retraces per frame. Zero is the fastest the computer can redraw. Compute FrameSpeed by dividing 60 by the number of frames per second you want and round to the next integer. Each frame will cause the NextFrame event to execute.

Starting and Stopping the Animation

To start the animation, add a Timer object to the window that contains the SpriteSurface and set its Mode to 2 (multiple calls). Its Period property specifies the time (in milliseconds between calls to the Timer. Set it to a reasonable value for your animation. In the Action event handler of the Timer, call the SpriteSurface's Update method.

The Update method runs the animation "one step at a time" — it redraws the sprites that have moved or changed, fires collision events, and calls the NextFrame event once.

To stop the animation, call the SpriteSurface's Stop method. Call the Close method to return the screen to the window that contains the SpriteSurface control.

You can also start the animation by simply calling the SpriteSurface's Run method, but this may lockup the user interface, making it impossible for the user to use the keyboard and mouse while the animation is running. Call the Update method from a Timer if you want to provide a modeless interface.

Sprite Surface Area

The SpriteSurface control has Left, Top, Width, and Height properties that are read/write. The SpriteSurface control can be resized in the Development

environment by dragging its handles with the mouse or by setting these properties. It can also be resized on-the-fly by resetting these property values.

To resize the SpriteSurface control so that it 'takes over' the user's monitor completely, use the FullScreen and MenuBarVisible properties of the parent window.

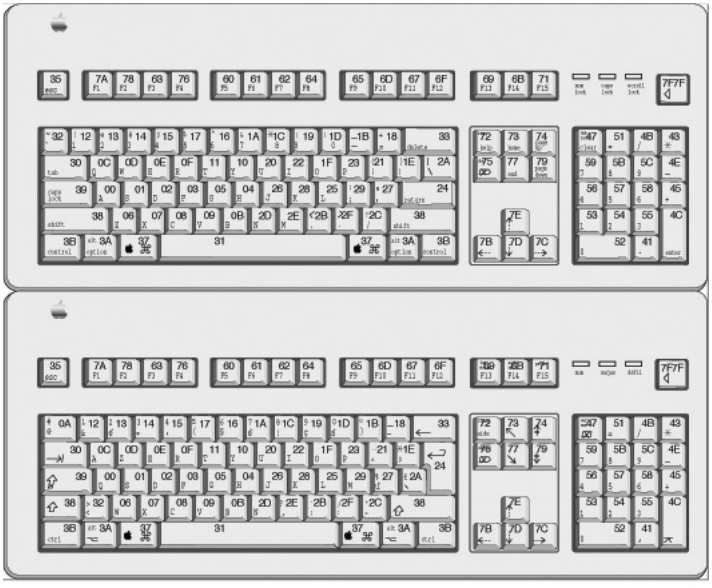
Set FullScreen to True and MenuBarVisible to False and use the SpriteSurface's Lock… properties to lock the SpriteSurface control to the parent window so that the SpriteSurface area expands automatically. This sets the Left, Top, Width, and Height properties accordingly.

The SpriteSurface Backdrop property can hold an image that is displayed when the animation begins. Because this is a picture, you can update it while the animation is running. However, you should avoid drawing into the active animation area. This area is controlled by the SurfaceWidth, SurfaceHeight, SurfaceLeft, and SurfaceTop properties.

Responding To The User During Sprite Animation

In the NextFrame event handler, you can use the SpriteSurface's KeyTest method to determine if the user is pressing a particular key. To use this method, you pass it a key code and the KeyTest method returns True if that key is being pressed and False if it's not. Key codes are not ASCII codes because some keys don't have ASCII codes (like the Shift, Command, and Option keys). Instead, key codes are special codes assigned to each key on the keyboard and they can vary for different keyboard configurations (i.e., between the English keyboard and the French keyboard). Figure 273 shows the English and French keyboards.

Figure 273. Keycodes for use with the KeyTest method.



Creating 3D Graphics Animations with the Rb3DSpace Control

You can use the Rb3DSpace control to create animations in a three-dimensional space. It works in conjunction with 3D classes which allow you to load, group, and manipulate 3D objects. The base 3D object is derived from the Object3D class; 3D objects are presented to the user via the Rb3DSpace control.

REALbasic's 3D capabilities are built on existing 3D libraries that provide a lot of power at very little cost. When you or your end-users run applications that use 3D animations, the appropriate libraries must be installed.

There are two independent libraries available: QuickDraw 3D, developed by Apple, and Quesa, an open-source library available on several platforms.

- Under "classic" Mac OS, you need QuickDraw 3D, which is installed by default. If it has somehow been removed, you can reinstall it using the QuickTime installer.
- Under Mac OS X, you need to install Quesa and OpenGL.
- Under Windows, you need to install Quesa and OpenGL.

Quesa is available from www.quesa.org. QuickTime is available from Apple Computer at www.apple.com/quicktime.

Since 3D animations are difficult to illustrate on the printed page, they are best illustrated by checking out the example 3D projects on the CD. The basic RB 3D Demo illustrates the following movements.

- **Yaw**: Left-Right keys
- **Pitch:** Up-Down keys
- **Zoom:** + and - keys moves the position of the "camera" in the 3D space.

The background and each of the creatures in the demo is a 3DMF object that is loaded into the Rb3DSpace. Animations are done using the keys shown in the right panel and are managed by the following code in the KeyDown event of the window:

```
Select case asc(Key)
  case 28 // left
    View3D1.Camera.Yaw 0.1
  case 29 // right
    View3D1.Camera.Yaw -0.1
  case 30 // up
    View3D1.Camera.Pitch -0.1
  case 31 // down
    View3D1.Camera.Pitch 0.1
  case 11 // page up
    View3D1.Camera.Roll -0.1
  case 12 // page down
    View3D1.Camera.Roll 0.1
  case 43 // keypad +
    View3D1.Camera.MoveForward 10.0
  case 45 // keypad -
    View3D1.Camera.MoveForward -10.0
  case asc("r")
    mPenguin.Yaw -0.1
  else
    Status.text = "Unknown key: " + str(asc(Key))
    return true
  end select


  View3D1.Refresh
```

View3D1 is the Rb3DSpace control and the Yaw, Pitch, Roll, and MoveForward methods of the Object3D class provide the animations as the user presses the indicated keys.

Please see the entry for the Rb3DSpace control in the Language Reference for information on how to go about placing objects in the 3D space and animating them.

CHAPTER 8

Working With Files

Many applications read from and/or write to files. Some create files that have their own special formats. Often this process starts with the user's selecting a file with the Open File dialog box or saving a file with the Save As dialog box. REALbasic makes it easy to use the Open and Save dialog boxes, as well as to read from and write to many different types of files.

Contents

Understanding File Types

There are many different file types. The type of a file defines a unique type of data stored in that file. For example, a text file stores text while a PICT file stores pictures. Under the Mac OS "classic", each file has a four-character File Type code and a four-character file Creator code stored with it. For Windows, Linux, and Mac OS X users, files have a file extension (or suffix) that defines the file type. For example, a Text file has the extension ".txt". A file named "myNotes.txt" is recognized as a Text file.

The file type makes it easy for an application to know if it is prepared to deal with a particular file. For example, any application that can open text files expects the file type of any text file it shall open is "TEXT". This file type tells the application that this is a standard text file. PICT files are so named because "PICT" is the file type of a PICT file. Applications are also files but all applications have a file type of "APPL" that tells the Mac OS that this file is executable and not just data.

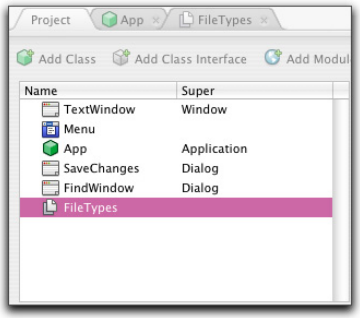
Rather than writing code that deals directly with all of these file types, creator codes, and file suffixes, REALbasic abstracts you and your code from them with *file types*. A file type in REALbasic is an item stored with your project that represents a specific file type, creator, and one or more extensions. Each file type has a name that is used in your code when opening and creating files. This allows you to work with names you can choose and easily remember instead of cryptic codes. It also abstracts your code from the operating system, making it easier for you to create versions of your application for other operating systems.

Using The File Types Editor

You use the File Type Sets Editor to create the items that will represent the different kinds of files you want your application to be able to open or create. To add file types to your project, you first add a File Type set to the IDE to hold your file types. The screen is called a File Type Set and it appears as an item in your Project Editor.

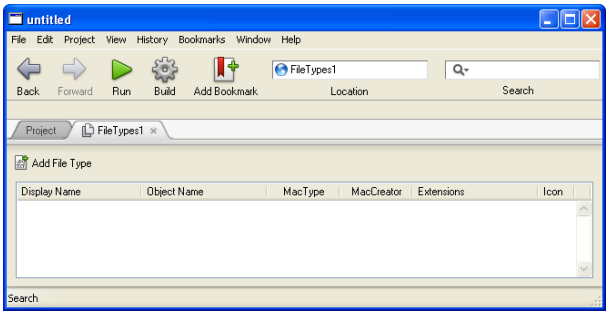
To add a File Types Set to your project, choose Project ▶ Add ▶ File Type Set. REALbasic adds a blank File Types Set to the project.

Figure 274. A File Types Set added to the Project Editor.



Double-click the FileTypes item to add file types to the File Types Set. The File Types Editor appears.

Figure 275. The File Types Set Editor.



The Display Name is the name shown to the user on some platforms in open file dialog boxes that indicates which file types can be opened.

The Object Name is the name used in your REALbasic code to identify the file type object. This is the string that you pass to a method to tell the method to use that file type.

The MacType and MacCreator are the Macintosh Type and Creator codes used by the original Macintosh operating system. If you are going to assign custom icons to the file type, be sure that the Creator code matches the Creator code that you give your application.

The Extensions are the three character file extensions used on Mac OS X, Windows, and Linux. You can specify more than one extension per file type. Separate multiple extensions with semicolons. The Icon is the document icon for that file type. When the user double-clicks such an icon, the standalone REALbasic application will start and open the file.

In the language, a File Type set has the string property "All" that returns all of the file types in the set. You can use All in your code when you want to refer to all of the file types in the set. Suppose you create a File Type Set called ImageTypes in which you specify all of the valid image types that your application can open. You can specify the entire list of image types with a line such as:

```
f=GetOpenFolderItem(ImageTypes.All)
```

If you need to add or remove image file types, your can simply modify the File Type Set and this line of code will automatically refer to the new members of the set.

Adding a File Type

To add a file type, do this:

1 Click a File Types Set item in the Project Editor.

A File Types Set Editor appears (Figure 275).

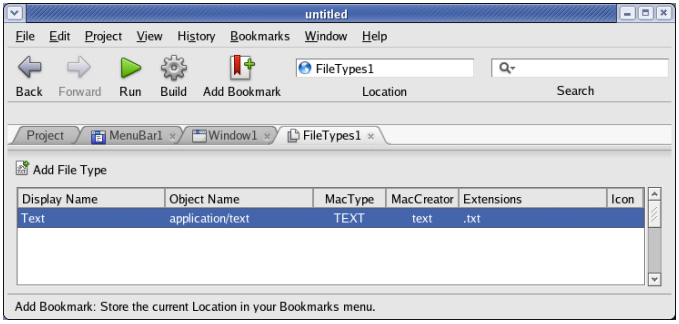
2 **Click the Add File Type button.**

A new row is added to the File Types table. Click twice in a cell to get an insertion point and press the Tab key to move from field to field.

3 **Enter the Display and Object names for the file type, the Mac Type and Creator codes, and the extensions.**

To enter more than one extension, separate them by semicolons.

Figure 276. A File Type added to the File Types Set table.



4 **If desired, repeat the process to add additional File Types to the set.**

5 **When you are finished with the File Types Set, click another tab or use the toolbar to move to another area of your project.**

Adding a Document Icon

You can define a custom document icon for the each file type that the application can open and save. REALbasic does not include an icon editor; you must design your icons in an icon editor program or image creation application and paste them into REALbasic.

You need to provide at least the following items:

- For Mac OSX, a 128 x 128 image, and for other platforms smaller sized icon images. Even for Mac OS X it is best to provide smaller sized image files at 32 x 32 and 16 x 16.
- A mask that defines the icon's edges so that the operating systems can determine which regions in the square image are clickable.

The document icons will be used when the application saves documents in that file type and double-clicking the document icon will start the application.

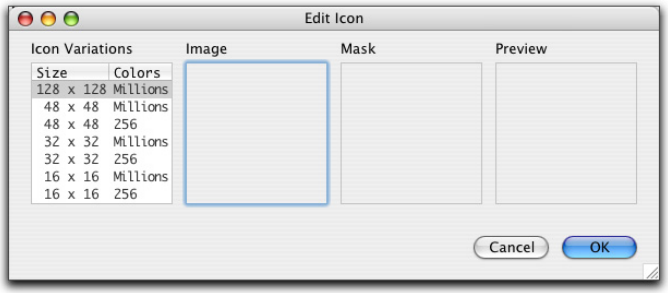
To copy and paste a document icon for the file type, do this:

1 **Place the icon you wish to paste into the Icon editor on the Clipboard.**

2 **Click the blank document icon in the Icon column for the file type.**

A Document icon dialog box appears, with entries for all sizes of document icons.

Figure 277. The Add Icon dialog box.



3 **In the Icon Variations list, select the icon size that matches the icon on the Clipboard and select the "Image" ImageWell.**

4 **Paste the icon on the Clipboard into the "Image".**

5 **Repeat this process for the icon's mask and, if needed, repeat the process for other icon sizes.**

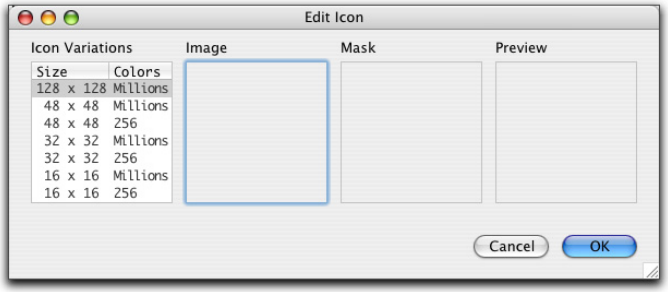
For Mac OS X you need to specify the 128 x 128 size icon and the OS will scale it as required.

To import a document icon for the file type, do this:

1 **Click the blank document icon in the Icon column for the file type.**

A Document icon dialog box appears, with entries for all sizes of document icons.

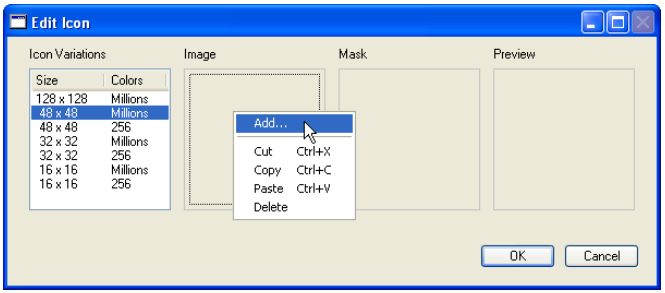
Figure 278. The Edit Icon dialog box.



2 **In the Icon Variations list, select the icon size that you want to import and select the "Image" ImageWell, as shown above.**

3 **Right+click (Ctrl+click on Macintosh) on the Image area to display its contextual menu and choose Add...**

Figure 279. The Add contextual menu.



An open-file dialog box appears.

4 **Choose the file that matches the selected icon size and click Open.**

On Windows, you can choose either a document in a picture format (e.g., bmp, jpg, gif) or a .ico file.

5 **Repeat this process for the icon's mask and, if needed, repeat the process for other icon sizes.**

For Mac OS X you need to specify the 128 x 128 size icon and the OS will scale it as required.

Editing a File Type

Making changes to file types is easy.

To edit a file type, do this:

1 **In the Project Editor, click the File Type Set you want to edit or click on its tab if it is already open.**

2 **Click on the file type in the File Type Set Editor you wish to edit to select it.**

3 **Click click twice in a cell in the row to get an insertion point.**

4 **Make any changes you wish and use the Tab key to move the insertion point to the next cell or click twice in another cell.**

5 **When you are finished, click another tab or use the toolbar to move to another area of your project.**

If you change the name of the file type, make sure you update any code that references the name of the file type. You can replace any occurrences of the old file type name with the new one easily using the Find/Change dialog box.

Deleting a File Type

Deleting a file type is simple.

To delete a file type, do this:

1 **In the Project Editor, click a File Type Set tab.**

2 **Click on the File Type you wish to edit to select it to select the whole row.**

3 **Press the Delete button on your keyboard.**

4 **When you are finished, click another tab or use the toolbar to move to another area of your project.**

If you delete a file type, make sure you update any code that uses this file type.

Using File Types

You pass one or more file types to commands to indicate that only the passed file types are appropriate. For example, the following statement in a control's Open event handler specifies that it can accept TEXT files that have been dragged from the Desktop:

```
me.AcceptFileDrop("text")
```

The statement

```
f=GetOpenFolderItem("video/quicktime")
```

displays an open-file dialog box that allows the user to view and open only QuickTime movies.

The FileType class has a built-in string conversion operator. This enables you to refer to the file type by its Object Name and it will return the appropriate string. For example:

```
f=GetOpenFolderItem(ImageTypes.JPEG)
```

would refer to the file type with the ObjectName of "JPEG" in the "ImageTypes" File Type Set.

You can concatenate two file type using the "+" operator to specify two file types. For example:

```
f=GetOpenFolderItem(ImageTypes.JPEG + ImageTypes.MacPICT)
```

refers to both the "JPEG" and "MacPICT" file types in the ImageTypes set.

The easiest way to specify several file types is to put all the file types that you want to refer to in one File Type Set and then use the All method of the File Type Set class. It automatically returns concatenates all the file types. For example, the following returns all the file types in the ImageTypes File Type Set.

```
f=GetOpenFolderItem(ImageTypes.All)
```

You can also specify more than one acceptable file type using their Object Names only. Separate the file type names by semicolons. For example, the following line allows the user to see and open PICT, GIF, and JPEG files:

```
f=GetOpenFolderItem("image/x-pict;image/jpeg;image/gif")
```

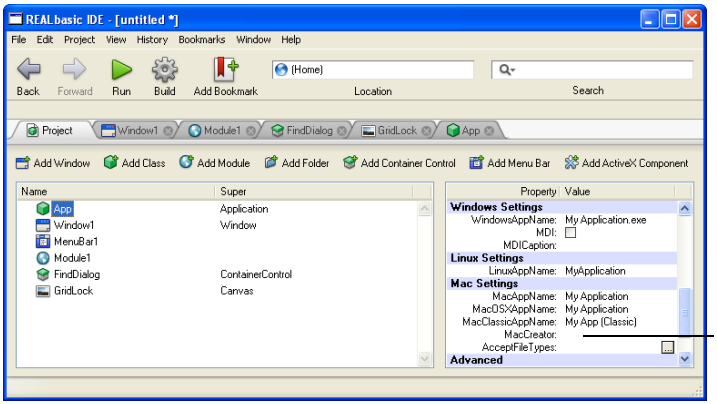
Consult the *Language Reference* for additional methods that take file types as arguments.

Creating Custom File Types for Your Application

Most applications create files and assign custom icons to them. These icons usually look similar to the application's custom icon. This makes it easier for the user to recognize that the file goes with the application that produced it. Any custom icons you add will appear only if you have assigned a Creator code to your project and built a stand-alone application.

To add custom icons to any of the file types for your project, first assign a unique Creator code to your application. You set the creator code in the Mac Settings group in the App class's Properties pane. It is shown in Figure 280.

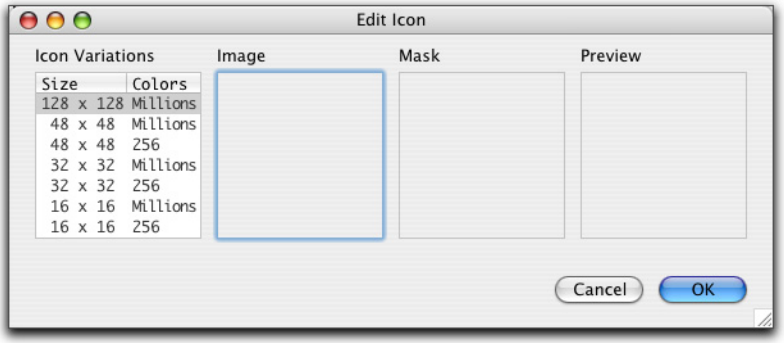
Figure 280. The Mac Settings group in the App class's Properties pane.



NOTE: Enter your Creator code into the Mac Creator area. Creator codes are case-sensitive.*Creator codes are case sensitive and must be unique. You can register a unique creator code for your application with Apple Computer at their web site at http://developer.apple.com/dev/cftype/find.html.*

You assign an application icon via the Icon property in the Appearance group. You need to design your icons in an external image or icon editing program. Click the "..." icon to display the Edit Icon dialog box and add your icons using the procedure described in the section "Adding a Document Icon" on page 376.

Figure 281. The Edit Icon dialog box.



Understanding FolderItems

To REALbasic, volumes, folders, applications, and documents are all considered to be *FolderItems*. A FolderItem is anything that can appear on the desktop. This doesn't mean that only items on the desktop are FolderItems. It means that if the item could be placed on the desktop, it's a FolderItem. For example, the Recycle Bin is a FolderItem because it appears on the desktop. See the section "Accessing Specific System Folders" on page 389 for more information on getting FolderItems for items that are created by the operating system.

The FolderItem class is your first point of contact with any item on a disk you want to read from or write to. To read from a file, for example, you get a FolderItem that represents the file, then use various methods to read from the file via the FolderItem. There are many different ways to get a FolderItem object that represents a particular volume, folder, application, or document. You can present the user with an Open File or Save As dialog box, you can get the FolderItem at a specific path, or you can even get a FolderItem from another FolderItem.

FolderItems have properties that store the path to the item, the name of the item, the size of the item, its type, etc. FolderItems also have methods you can use to create files, open files, delete files, copy files, etc.

For detailed information on the properties and methods of the FolderItem Class, see the FolderItem class in the *Language Reference*.

How Are Shortcuts and Aliases Handled?

Shortcuts (aliases in Macintosh terminology) are files that actually represent a volume, application, folder, or file stored in another location and possibly under another name. REALbasic contains commands that allow you to either resolve the shortcut and work with the actual object or work with the object directly. The GetFolderItem function automatically resolves an shortcut when it encounters it, while the GetTrueFolderItem function works with the shortcut itself.

Getting a File at a Specific Location

If you know the full path to a file and you wish to access the file, you can do so by specifying the path to the file.

For example, suppose you have a document called “Schedule” stored in the same folder as the REALbasic application.The relative path starts with the directory REALbasic (or the REALbasic application) is in. If the file is in the same directory as REALbasic, then the relative path consists only of the filename itself.

The following code creates a FolderItem object in the local variable “f” that represents the document mentioned above:

```
Dim f as FolderItem
f=GetFolderItem("Schedule")
```

The GetFolderItem function should be used to either specify a file or directory in the same directory as the application or the empty string (""). If you specify the latter, GetFolderItem will return the FolderItem for the directory in which your application is located.

```
Dim f as FolderItem
f=GetFolderItem("") //returns the directory containing your application
```

The full path (sometimes called the absolute path) to a volume, directory, application, or document starts with the volume name followed by the path delimiter character (a backslash on Windows, a colon on the Macintosh, and a forward slash on Linux), the names of any folders in the path (each separated by the path delimiter) and ending with the name of the item.

To create an absolute path to a file or folder, you should use the Volume global function to build a full path to the item, starting with the hard disk it is on. You then use the Child method of the FolderItem class to navigate to the item. Volume returns a FolderItem for one of your mounted volumes. You specify the volume by passing an integer, indicating the volume. Volume 0 is the volume that contains the operating system — the “boot” volume.

```
Dim f as FolderItem
f=Volume(0) //returns a folderitem for the boot volume
```

Parent returns the FolderItem for the next item up in the absolute path for the current FolderItem. It does not work if you try to get the parent of a volume. The Child method lets you access any items one level below the current FolderItem.

You can build a full path starting from a volume with the Child method. For example, if you want to get a FolderItem for the file "Schedule" in the directory "Stuff" on the boot volume, the statement would be:

```
Dim f as FolderItem
f=Volume(0).Child("Stuff").Child("Schedule")
```

The following example works with a relative path. It uses the Parent property to get the FolderItem for the directory that contains the directory in which the application is located. Passing the empty string to GetFolderItem gets the current directory, so the parent of that directory is one level up in the hierarchy.

```
Dim f as Folderitem
f=GetFolderItem("").Parent
```

Once you have a FolderItem, you can (depending on what type of item it is) copy it, delete it, rename it, read from it or write to it, etc. You will learn how to read and write to files using FolderItems later in this chapter.

Verifying that you have accessed the Item

When you try to get a FolderItem, either of two things can go wrong. First, the path may be invalid. An invalid path contains a volume reference and/or a directory name that doesn't even exist. For example, if you specify Volume (1) when the user has only one volume, the Volume function returns a Nil value in the FolderItem instance, f. If you try to use any of the FolderItem class's properties or methods on a Nil FolderItem, a NilObjectException error will occur. If the exception is not handled in some way, the application will shut down.

Second, the path may be valid, but the file you are trying to access may not exist. The following shell code checks for these two situations:

```
Dim f as FolderItem
f=Volume(0).Child("Documents").Child("Schedule")
If f <> Nil then
  If f.Exists then
    //proceed to access the folderitem
  Else
    MsgBox "The document does not exist!"
  End if
Else
  MsgBox "You supplied an invalid path!"
End if
```

If the path is valid, the code checks the Exists property of the FolderItem to be sure that the file already exists; if the file doesn't exist or the path is invalid, a warning message is displayed.

You can also handle an invalid path using an Exception Block. They are discussed in the section "Runtime Exception Errors" on page 510.

Creating a new FolderItem

You can create a FolderItem for an existing item by passing it the pathname. When you create a FolderItem with the New command, you can pass the path to the new FolderItem as an optional parameter. For example:

```
Dim f as FolderItem
f=New FolderItem("myDoc.txt")
```

specifies the name of the new FolderItem and it is located in the same folder as the REALbasic application (if you're running in the IDE) or the same folder as the built application.

If you pass a FolderItem instead of a path, the New method will create a copy of the passed FolderItem. In this example, the FolderItem "f2" refers to a copy of the original FolderItem, not a reference to it.

```
Dim f, f2 as FolderItem
f=Volume(0).Child("Documents").Child("Schedule")
f2 = New FolderItem(f)
```

Getting Information About a FolderItem

If GetFolderItem returns a valid FolderItem to an existing item, you can get information about the FolderItem using the local variable "f". For example, you can get the modification date of the FolderItem. This example displays the modification date of the FolderItem above:

```
Dim f as FolderItem
f=Volume(0).Child("Documents").Child("Schedule")
if f <> Nil then
  if f.Exists then
    MsgBox f.ModificationDate.ShortDate
  End if
End if
```

Deleting a FolderItem

Once you have a FolderItem that represents an item that can be deleted, you can call the FolderItem's Delete method. The following example deletes the file represented by the FolderItem returned:

```
Dim f as FolderItem
f=Volume(0).Child("Documents").Child("Schedule")
If f <> nil Then
  if f.Exists then
    f.Delete
  End if
End if
```

If the FolderItem is locked, an error will occur. You can check to see if the FolderItem is locked by checking the FolderItem's Locked property.

Deleting a FolderItem does not simply move the FolderItem to the trash. The FolderItem is deleted permanently from the volume.

Getting and Setting Ownership

On Unix-based operating systems, permissions to read, write to, and run a file (or view the contents of a directory) are regulated by a system of permissions. Central to permissions system is file/directory ownership. Each file or directory has an Owner and the Owner is in a Group.

The FolderItem's Owner and Group properties let you read and set this information. For example, if we use the previous example, we can get the name of the Owner of the document in the following way:

```
Dim f as FolderItem
f=Volume(0).Child("Documents").Child("Schedule")
if f <> Nil then
  if f.Exists then
    MsgBox "Schedule owner is "+f.Owner
  End if
End if
```

To get the name of the owning Group, read the value of f.Group instead of f.Owner. You can assign a new owner or group simply with assignment statements, such as:

```
f.Owner="Matt"
f.Group="Marketing"
```

Getting and Setting Permissions

Unix-based operating systems (Mac OS X and Linux), use a system of permissions to regulate access to files and directories. Permissions is granted to the Owner of the item, members of the owning Group of users, and all other users not in the owning Group.

Three levels of permissions may be granted:

- **Read:** A user may only read the file or view the name of a directory.
- **Write:** A user may change the file or directory.
- **Execute:** A user may run the file (application) or view the contents of a directory and examine the files.

Under Unix, permissions is indicated in a rather cryptic way. It uses a three-digit octal number in which each digit indicates the permissions for the Owner, Group, and Others, in that order from left to right.

The code for each digit is computed using the following values:

Table 30: Values for Read, Write, and Execute Permissions

Value	Permission Type	Description
4	Read permissions	User or group can open and read the FolderItem.

Table 30: Values for Read, Write, and Execute Permissions

Value	Permission Type	Description
2	Write permissions	User or group can open and write to the FolderItem.
1	Execute permissions	User or group can execute the file or read the directory.

For each digit, the permissions are expressed by adding up the values. Each digit can take on eight possible values. Since each digit can only take on one of 8 values, its an *octal* system, or Base 8.

Table 31: All possible values for a permissions integer.

Value	Description
0	No permissions
1	Execute permissions
2	Write permissions
3	Write and Execute permissions
4	Read permissions
5	Read and Execute permissions
6	Read and Write permissions
7	Read, Write, and Execute permissions

For example, a permissions of 664 means that the Owner and Group (who each have a value of 6 in this example) have Read and Write permissions, but Others have only Read permissions.

The FolderItem's Permissions property lets you get and set permissions by via this octal value. You can read the value or just set it to a new value by assigning this property the three-digit octal value that sets the desired permissions.

You can also get or set permissions for a FolderItem via the Permissions class. This class lets you avoid having to deal with octal numbers and their meaning and, instead indicates permissions by a group of Boolean properties. Each property is True if the user has the specified permissions.

The Boolean properties are as follows.

Table 32: The Permissions class's properties for setting access permissions.

Name	Description
OwnerExecute	If True, the owner of the FolderItem can execute the FolderItem.
OwnerRead	If True, the owner of the FolderItem can read the FolderItem.
OwnerWrite	If True, the owner of the FolderItem can write to the FolderItem.
GroupExecute	If True, a member of the Group that owns the FolderItem can execute the FolderItem.
GroupRead	If True, a member of the Group that owns the FolderItem can read the FolderItem.

Table 32: The Permissions class's properties for setting access permissions.

Name	Description
GroupWrite	If True, a member of the Group that owns the FolderItem can write to the FolderItem.
OtherExecute	If True, a user outside the Group that owns the FolderItem can execute the FolderItem.
OtherRead	If True, a user outside the Group that owns the FolderItem can read the FolderItem.
OtherWrite	If True, a user outside the Group that owns the FolderItem can write to the FolderItem.

In addition, the Permissions class gives you access to the Set User Bit and Set Group Bit. These Boolean flags grant the current user the permissions of the Owner and Group, respectively, without actually changing the FolderItem's permissions. These properties are shown inTable 33.

Table 33: The Set User Bit and Set Group Bit properties.

Name	Description
GIDBit	Set Group ID bit. If True, the current user has the permissions of the owning Group without altering the Read/Write/Execute permissions for the FolderItem.
UIDBit	Set User ID bit. If True, the current user has the permissions of the owner without altering the Read/Write/Execute permissions for the FolderItem.

You can also use the Permissions class to get or set the so-called "Sticky" bit of a FolderItem. If the Sticky bit is set, the operating system will not allow someone to remove or rename the file or files in the directory that he does not own, even he has Read/Write permissions. This can be used for directories in which different users need to be able to create files but should not touch others' files.

All nine bits of information are shown in the Linux version of the Properties/Get Info dialog box for a file or directory. This is shown in Figure 282. The Mac OS X Get Info dialog displays only Read and Write permissions.

Figure 282. A Linux Properties dialog box for a file.

The values of the nine properties shown in Table 32 are shown via the nine CheckBoxes in the top section of the dialog. The Set User, Set Group, and Sticky Bit settings are displayed in the Special Flags section. Notice that the Number View (near the bottom of the dialog) shows the User, Group, and Others permissions in Octal form. Since the User and Group have Read, Write, and Execute permissions for this file, the value is 7. Others have only Read and Execute permissions, so the value is 5.

To use the Permissions class, you create an instance of the class by passing it the value of the Permissions property of the FolderItem. For example,

```
Dim f as FolderItem
f=GetOpenFolderItem("")
if f <> Nil then
  if f.Exists then
    Dim perms as New Permissions(f.Permissions)
    //get or set permissions class bits here...
    //with the properties in Table 32 and Table 33
  End if
End if
```

Getting The Path To Your Application's Folder

Passing a null string (two quotes with no characters in between them) to the GetFolderItem function returns a FolderItem representing the folder your application or project is in. You can then use the FolderItem's Item method to access all the items in the folder your application is in. The Item method returns an array of FolderItems in the directory.

Getting Specific Items In the Application's Folder

If the first item in the path is not a volume, the GetFolderItem function assumes that the first item in the path is in the same folder as the application. If you are running your project in REALbasic, GetFolderItem looks for the item in the folder your project is in. If you haven't saved your project yet, GetFolderItem will look in the folder that REALbasic is in.

The following example returns a FolderItem that represents a file called "My Template" in a folder called "Templates" that is located in the same folder as the application:

```
Dim f as FolderItem
f=GetFolderItem("").Child("Templates").Child("My Template")
```

If the file is in the same folder as the application, then you can just pass GetFolderItem the name of the file. The following works:

```
Dim f as FolderItem
f=GetFolderItem("My Template")
```

Accessing Specific System Folders

REALbasic provides a module that returns FolderItems representing various folders that are part of the System software or the desktop. When you need to access one of these folders, you should use the SpecialFolder object to obtain the FolderItem. These functions will still work properly even if the directory's name changes. They are also language independent. For more information on these functions, see the *Language Reference*.

You obtain the desired FolderItem using the syntax:

```
result=SpecialFolder.FolderName
```

where *result* is the FolderItem you want to obtain and *FolderName* is the name of the SpecialFolder function that returns that FolderItem. For example, the following gets a FolderItem for the Application Support folder on Mac OS X and the Application Data directory on Windows.

```
Dim f as FolderItem
f=SpecialFolder.ApplicationSupport
```

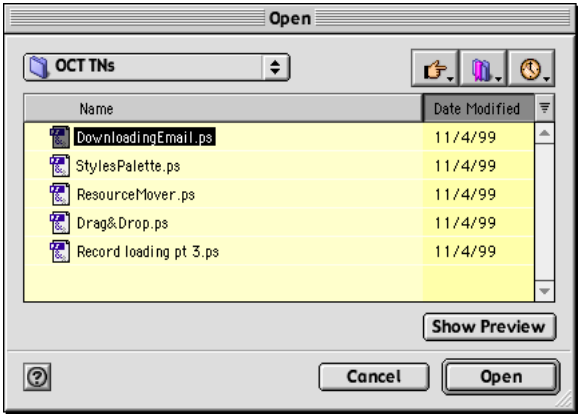
The entry for SpecialFolder in the *Language Reference* has the complete list of supported functions and the FolderItems that they return on Mac OS X, Windows, and Linux. Not all functions return FolderItems on all platforms. Please refer to that list.

Getting The Selected File From An Open File Dialog Box

The Open File dialog box lets the user navigate to a particular location on any mounted volume and select a file to open. Figure 283 shows an example of the Mac OS 9 Open File dialog box.

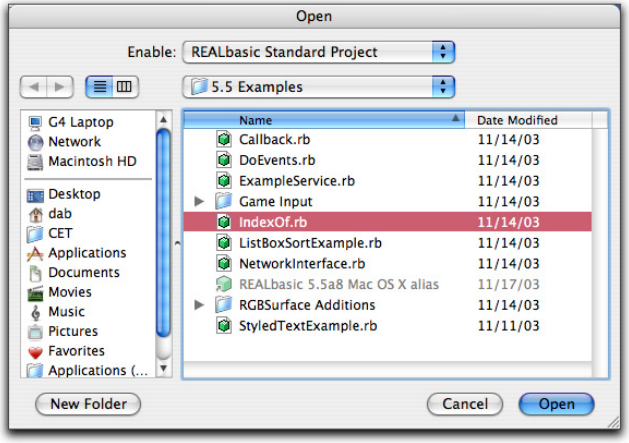
If the user is running Mac OS 8.5-9.2, the Navigation Services open-file dialog box appears.

Figure 283. The Navigation Services Open File dialog box.



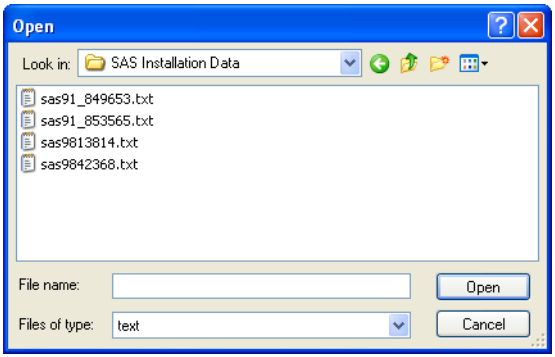
If the user is running Mac OS X 10.3 or above, the following dialog box appears:

Figure 284. The Mac OS X file browser (10.3).



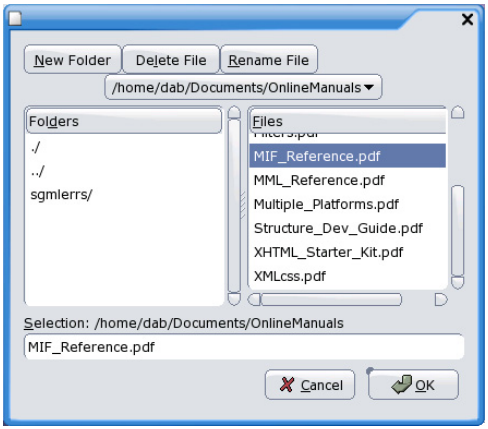
The Windows open file dialog box is shown in Figure 285 on page 391.

Figure 285. The Windows Open FolderItem dialog box.



A Linux version of the dialog box is shown below.

Figure 286. The Linux Open FolderItem dialog box.



To present the user with an Open File dialog box, you can call either the GetOpenFolderItem function or use the OpenDialog class.The former is a standard function that presents a standard open-file dialog box.The latter allows you to create a customizable open-file dialog box in which you can specify the following aspects of the dialog:

- Position (Left and Top properties)
- Default directory (Initial Directory property)
- Valid file types to show (Filter property)
- Text of Validate and Cancel buttons (ActionButtonCaption and CancelButtonCaption properties)
- Text that appears above the file browser (Title property)
- Text that appears below the file browser (PromptText property)

The GetOpenFolderItem function displays the Open File dialog box and returns a FolderItem object that represents the file the user selected. One or more file types (that have been defined in the File Type Sets Editor or with the FileType class via the language.) must be passed to the GetOpenFolderItem function. It presents only those file types to the user in its browser. In this way, the user can only open files of the appropriate type. To pass more than one file type, separate them with semicolons.

The following example displays the Open File dialog box, allowing the user to select only jpeg or PICT files, and then displays the selected file's modification date:

```
Dim f as FolderItem
f=GetopenFolderItem("image/jpeg;image/x-pict")
MsgBox f.ModificationDate.ShortDate
```

If the user clicks the Cancel button rather than the Open button in the Open File dialog box, GetOpenFolderItem returns Nil. You will need to make sure the value returned is not Nil before using it. If you don't, a NilObjectException error will be generated. The following example shows how the code from the previous example should be written to check for a Nil object:

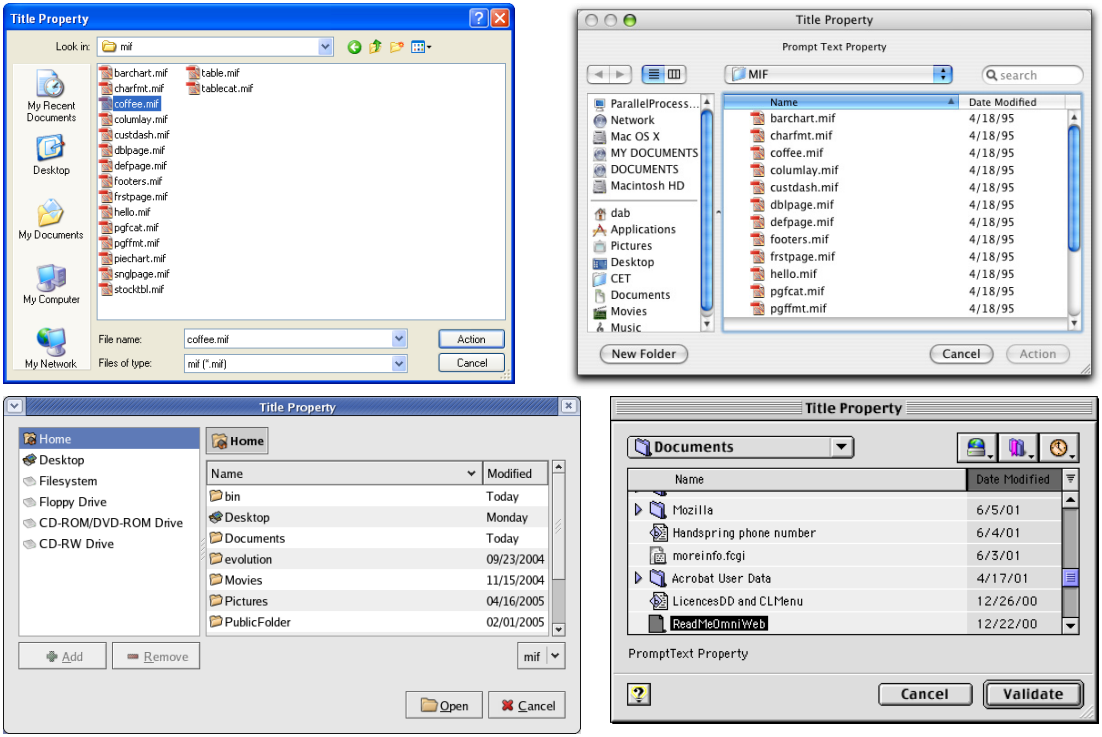
```
Dim f as FolderItem
f=GetOpenFolderItem("image/jpeg;image/x-pict")
If f <> Nil Then
 MsgBox f.ModificationDate.ShortDate
End if
```

When you use the OpenDialog class, you create a new object based on this class and assign values to its properties to customize its appearance. The following example uses a custom prompt and displays only one file type:

```
Dim dlg as OpenDialog
Dim f as FolderItem
dlg=New OpenDialog
dlg.InitialDirectory=Volume(0).Child("Documents")
dlg.Title="Select a MIF file"
dlg.Filter="application/x-mif"
f=dlg.ShowModal()
If f <> Nil then
 //proceed normally
 Else
//user cancelled
End if
```

Customized open-file dialog boxes are shown in Figure 287.

Figure 287. OpenDialog boxes on Windows, Mac OS X, Linux and Mac OS "classic.".



For more information, see the GetOpenFolderItem class and the OpenDialog class in the *Language Reference*.

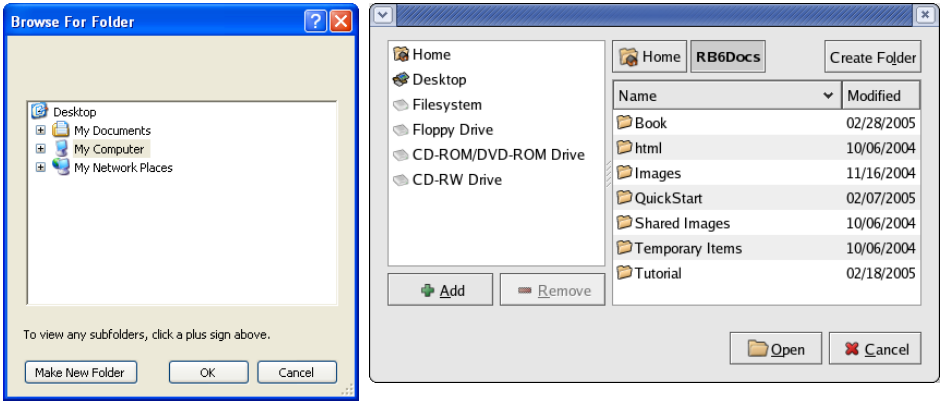
For more information on file types, See "Understanding File Types" on page 374.

Getting The Selected Folder From An Open Folder Dialog Box

The Open File dialog box doesn't allow the user to select a folder. Fortunately, REALbasic's SelectFolder function displays an Open Folder dialog box that lets the user choose a folder rather than a file. The SelectFolderDialog class performs the same function, but allows you to customize the appearance of the dialog.

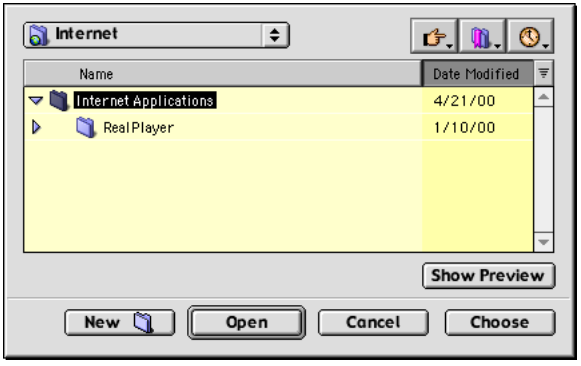
On Windows and Linux, the SelectFolder function displays the following dialog box:

Figure 288. The Windows and Linux SelectFolder dialog boxes.



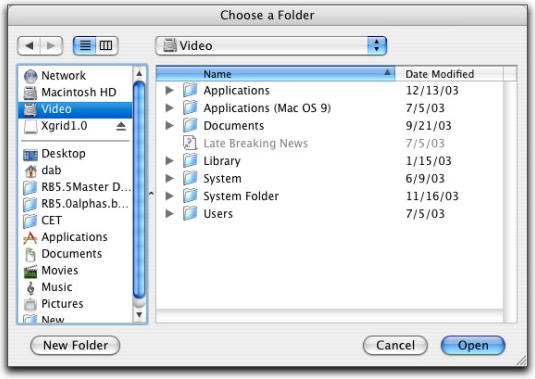
If the user is running MacOS8.5-9.2, the Navigation Service's Open Folder dialog box appears.

Figure 289. The Navigation Services Open Folder dialog box.



If the user is running Mac OS X (10.3 or above), the browser shown in Figure 290 appears.

Figure 290. The Mac OS X (10.3 or above) Select Folder dialog.



The SelectFolder function returns a FolderItem that represents the folder the user selects when he clicks the Open button at the bottom of the dialog box (or the Choose button in the Navigation Services Open Folder dialog box). If the user clicks the Cancel button rather than the Select/Choose button, SelectFolder returns Nil. You need to check for it before using the returned value.

The following example displays the number of items in the folder selected by the user:

```
Dim f as FolderItem
f=SelectFolder
If f <> Nil Then
 MsgBox Str(f.Count)
End if
```

When you use the SelectFolderDialog class, you create an object based on this class and assign values to its properties to customize its appearance. You can customize the following properties:

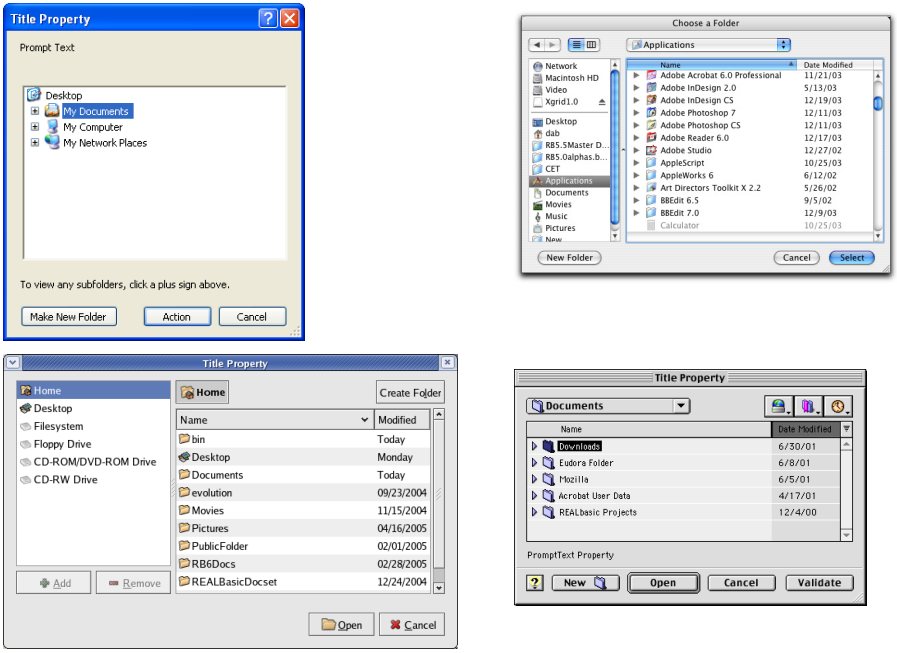
- Position (Left and Top properties)
- Default directory (Initial Directory property)
- Valid file types to show (Filter property)
- Text of Validate and Cancel buttons (ActionButtonCaption and CancelButtonCaption properties).
- Text that appears above the file browser (Title property)
- Text that appears below the file browser (PromptText property)

The following example opens a select folder dialog box and presents the contents of the Applications folder on the user's startup volume in the browser:

```
Dim dlg as SelectFolderDialog
Dim f as FolderItem
dlg=New SelectFolderDialog
dlg.ActionButtonCaption="Select"
dlg.InitialDirectory=Volume(0).Child("Applications")
f=dlg.ShowModal()
if f <> Nil then
 //use the folderitem here
else
 //user cancelled
end if
```

Select folder dialog boxes created with the SelectFolderDialog class are shown in Figure 291.

Figure 291. Select folder dialogs on Mac OS, Mac OS X, and Windows.



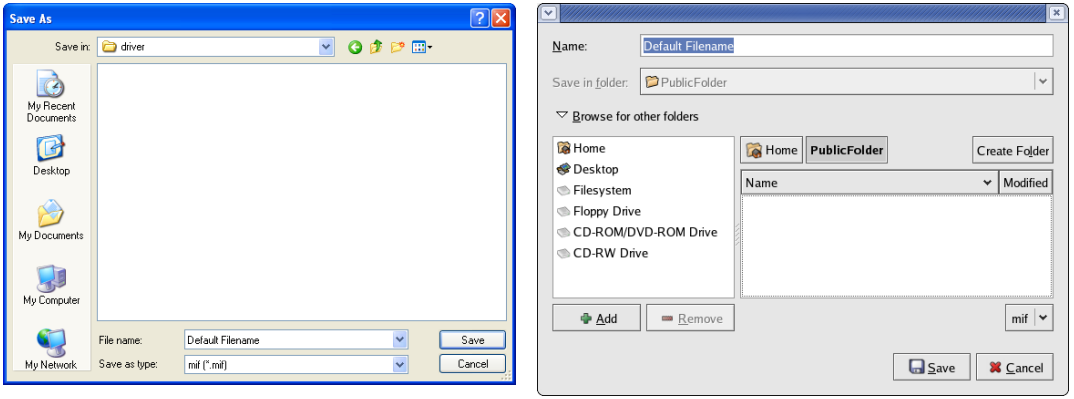
For more information, see the SelectFolder and SelectFolderDialog classes in the *Language Reference*.

Using the Save As Dialog Box

The Save As dialog box is used to let the user choose a location in which to save a file and give the file to be saved a name.

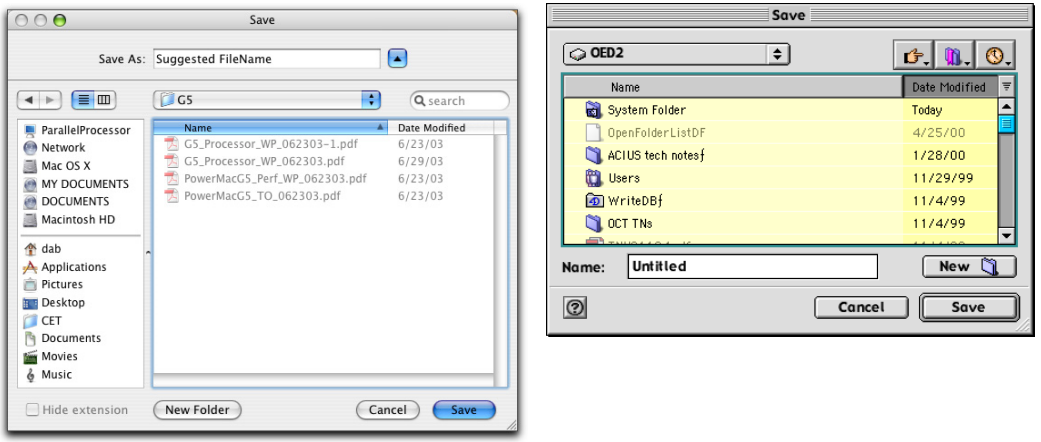
The Windows and Linux versions of this dialog box are shown in Figure 292.

Figure 292. The Windows and Linux Save As dialog boxes.



The versions of this dialog for Mac OS X and "classic" are shown below.

Figure 293. The Mac OS X and Mac OS "classic" Save As dialog boxes.



REALbasic's GetSaveFolderItem function presents the Save As dialog box. The SaveAsDialog class allows you to create a customized version of this dialog. Both objects return a FolderItem that represents the file the user wishes to save. This is an important distinction because the file doesn't exist yet. You must provide additional code that will create the file and write the data to the file. You will learn about creating files and writing data later in this chapter.

When you call the GetSaveFolderItem function, you define the type of file and the default name for the file (that will appear in the Name field in the Save As dialog box). The file type (which is the first parameter of the function) is the name of any file type defined for the project in the File Types dialog box.

Like the other functions that return FolderItems, you should make sure the FolderItem returned by GetSaveFolderItem is not Nil before using it (The FolderItem will be Nil if the user clicked Cancel).

The following example presents the Save As dialog box. The dialog presents a default file name of "Untitled". It also returns a FolderItem whose Type and Creator match the "pict" file type as defined for the project in the File Types dialog box, in a File Types Set called "FileTypes1". If the user clicks the Save button, the name the user chose for the file is displayed:

```
Dim f as FolderItem
f=GetSaveFolderItem(FileTypes1.pict,"Untitled")
If f <> Nil and f.Exists Then
  MsgBox f.name
End if
```

NOTE: If you are going to create a text file with the FolderItem returned, you can pass an empty string as the first parameter of the GetSaveFolderItem function. The method that creates a text file (CreateTextFile) will assign the file type and creator automatically.

When you use the SaveAsDialog class, you create a new object based on this class and customize the dialog by assigning values to its properties. You can customize the following aspects of the dialog:

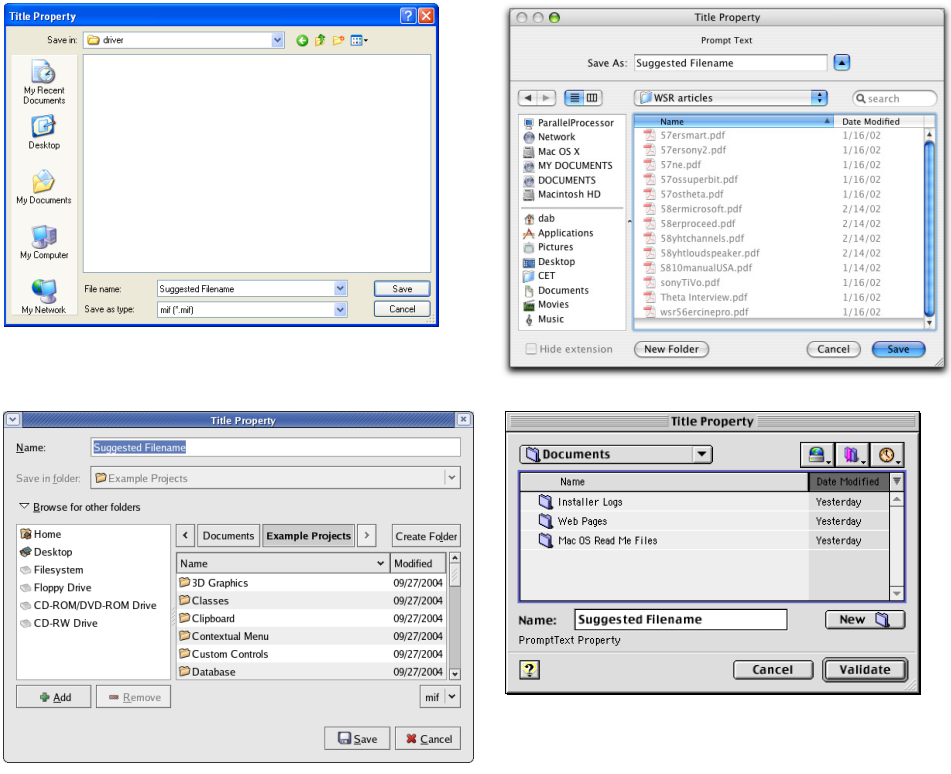
- Position (Left and Top properties)
- Default directory (Initial Directory property)
- Valid file types to show (Filter property)
- Default filename (SuggestedFileName property)
- Text of the Validate and Cancel buttons (ActionButtonCaption and CancelButtonCaption properties)
- Text that appears above the file browser (Title property)
- Text that appears below the file browser (PromptText property)

The following example opens a customized save-file dialog box and displays the contents of the Documents directory in the browser area.

```
Dim dlg as SaveAsDialog
Dim f as FolderItem
dlg=New SaveAsDialog
dlg.InitialDirectory=Volume(0).Child("Documents")
dlg.promptText="Enter a filename"
dlg.Filter=FileTypes1.rtf  //rtf type defined in FileTypes1 set
dlg.SuggestedFileName="RTF file"
dlg.Title="Save your file"
f=dlg.ShowModal()
If f <> Nil then
 //file saved
Else
//user canceled
End if
```

Customized save-as dialog boxes created using the SaveAsDialog class are shown in Figure 294.

Figure 294. Save as dialogs on Windows, Mac OS X, and Linux, and Mac OS "classic".



For more information on file types, See "Understanding File Types" on page 374.

For more information, see the GetSaveFolderItem function and the SaveAsDialog class in the Language Reference.

Working With Text Files

Text files can be read by text editors (like SimpleText, gedit, and BBEdit) and word processors (like Microsoft Word). Text files can easily be created, read from, or written to with REALbasic. Text files are convenient since they can be read by many other applications.

Whether you are going to read from a text file or write to a text file, you must first have a FolderItem that represents the file you are going to read from or write to.

Reading From a Text File

Once you have a FolderItem that represents an existing text file you wish to open, you open the file using the OpenAsTextFile method of the FolderItem class. This method is a function that returns a "stream" that carries the text from the text file to your application. The stream is called a *TextInputStream*. This is a special class of object designed specifically for reading text from text files. You then use ReadAll or ReadLine methods of the TextInputStream to get the text from the text file. The TextInputStream keeps track of the last position in the file you read from.

The ReadAll method returns all the text from the file (via the TextInputStream) as a string. The ReadLine method returns the next line of text (the text after the last character read but before the next end of line character). As you read text, you can determine if you have reached the end of the file by checking the TextInputStream's EOF (end of file) property. This property will be True once the end of the file is reached. When you are finished reading text from the file, call the TextInputStream's Close method to close the stream to the file, making the file available to be opened again.

This example lets the user choose a text file using the Open-file dialog box and displays the text in an EditField. It assumes that the valid text file types have been defined in a File Type Set called TextTypes:

```
Dim f as FolderItem
Dim stream as TextInputStream
f=GetOpenFolderItem(TextTypes.All) //all the file types in this set
If f<> Nil Then
  stream=f.OpenAsTextFile
  EditField1.text=stream.ReadAll()
  stream.Close
End if
```

NOTE: Because ReadAll reads all of the text in the file, the resulting string will be as large as the file. Keep this in mind because reading a large file could require more memory than the user has available for the application.

This example reads the lines of text from a file stored in the Preferences folder in the System folder into a ListBox.

```
Dim f as FolderItem
Dim stream as TextInputStream
f = SpecialFolder.Preferences.child("My Apps Prefs")
If f <> Nil and f.Exists then
  stream = f.OpenAsTextFile
  While Not stream.EOF
    ListBox1.addrow stream.ReadLine
  Wend
  stream.Close
End if
```

Specifying an Encoding

If you are reading and writing text files with only your REALbasic application, this code will work. However, if the files are coming from other applications or platforms, in other languages or a mixture of languages, then you may need to specify the encoding of the text. This is because the character codes above ACSII 127 may differ from what your application expects. When you read text, you can set the Encoding property of the TextInputStream to the encoding of the text file.

Here is the first example, amended to specify the text encoding of the incoming text stream. The code assumes that there is a File Type Set in the application named TextTypes.

```
Dim f as FolderItem
Dim stream as TextInputStream
f=GetOpenFolderItem(TextTypes.All)
If f<> Nil Then
stream=f.OpenAsTextFile
  Stream.Encoding=Encodings.Windows.ANSI //specify the ANSI encoding
  EditField1.text=stream.ReadAll()
  stream.Close
End if
```

The Encodings object provides access to all encodings. Use it whenever you need to specify an encoding. You can also specify the text encoding by passing the encoding as an optional parameter to Read or ReadAll.

For more information about text encodings, see the section "Working with Text Encodings" on page 333.

Writing to a Text File

Once you have a FolderItem that represents the text file you wish to open and write to, you open the file using the AppendToTextFile method of the FolderItem class. If you are creating a new text file or overwriting an existing text file, use the CreateTextFile method of the FolderItem class. These methods are functions that return a "stream" that carries the text from your application to the text file. The

stream is called a *TextOutputStream*. This is a special class of object designed specifically for writing text to text files. You then use the WriteLine method of the TextOutputStream class to write the text to the text file. Text written to a text file is always appended to the end of the text file.

The WriteLine method, by default, adds a carriage return to the end of each line. This is controlled by the TextOutputstream's Delimiter property which can be changed to any other character.

When you are finished writing text to the file, call the TextOutputStream's Close method to close the stream to the file making the file available to be opened again.

This example displays the Save As dialog box then writes the contents of three EditFields to the text file and closes the stream. It assumes that there is a file type called "text" in the TextTypes File Type Set.

```
Dim file As FolderItem
Dim fileStream As TextOutputStream
file=GetSaveFolderItem(TextTypes.text,"My Info")
fileStream=file.CreateTextFile
fileStream.WriteLine namefield.Text
fileStream.WriteLine addressfield.Text
fileStream.WriteLine phonefield.Text
fileStream.Close
```

Specifying an Encoding

As is the case with reading text files, you may need to specify an encoding when you write out a text file. If the application that will read the file is expecting that the text is in a specific encoding, you should convert the text to that encoding before exporting it.

Before writing out a line or the entire block of text (with the Write method) use the ConvertEncoding function to convert the encoding of the text. Here is a revised example. It converts the text to the MacRoman encoding.

```
Dim file As FolderItem
Dim fileStream As TextOutputStream
file=GetSaveFolderItem(TextTypes.text,"My Info")
fileStream=file.CreateTextFile
fileStream.Write ConvertEncoding(namefield.Text,Encodings.MacRoman)
fileStream.Close
```

Limitations of Text Files

Text files can only be accessed sequentially. This means that to read some text that is in the middle of the file, you must read all of the text that comes before it. It also means that to write some text to the middle of a text file, you have to write all of the text that comes before the text you wish to insert, then write the text you wish to insert, then the text that follows the text you wish to insert. You can not read text from a text file and write to the same text file at the same time. If these limitations

are going to be a problem for your project, consider using a binary file instead. For more information on binary files, See "Working With Binary Files" on page 408.

Working With Styled Text Files

REALbasic makes it easy to read from and write to text files that support styled text. SimpleText is an example of an application that supports styled text.

Loading Styled Text Into an EditField

Once you have a FolderItem that represents the styled text file you wish to read text from, you can read the styled text using the OpenStyledEditField method of the FolderItem class. To use this method, pass it the EditField you wish to display the styled text in. This EditField must have its Styled property set to True.

This example displays an Open File dialog box. It then reads the styled text from the file chosen and displays it in an EditField. It assumes that all the text file types that you want to read are defined in a File Type Set called "TextTypes".

```
Dim f as FolderItem
f=GetOpenFolderItem(TextTypes.All)
If f <> Nil Then
 f.OpenStyledEditField EditField1
End if
```

Writing Styled Text From an EditField to a File

Once you have a FolderItem that represents the styled text file to which you wish to open and write to, you can write the styled text using the SaveStyledEditField method of the FolderItem class. To use this method, pass it the EditField from which you wish to get the styled text. This EditField must have its Styled and MultiLine properties set to True.

This example displays the Save As dialog box. It then writes the styled text from EditField1 to a new file. It uses the Text file type as defined in the File Type Set "TextTypes".

```
Dim f as FolderItem
f=GetSaveFolderItem(TextTypes.Text,"Untitled")
If f <> Nil Then
 f.SaveStyledEditField EditField1
End if
```

Working with StyledText Objects

The other way to save and load styled text is to use the StyledText class. It enables you to work with styled text that is not connected to an EditField. You can manipulate the styles and paragraph alignments within the StyledText object and save it to disk for later use.

You work with StyledText as a series of concatenated StyleRun objects. The StyleRun entry in the Language Reference shows how to save a StyledText object to disk using a BinaryStream. The approach is to save the sequence of StyleRuns into a

MemoryBlock object and then write the MemoryBlock to disk by calling the Write method of the BinaryStream class. See the entry for StyleRun for the sample code that does this.

Working With Picture Files

REALbasic has built-in support for opening and saving both bitmap and vector picture files. On Macintosh, it supports vector and raster PICT files and on Windows, it supports the .bmp (bitmap) and .emf (Extended Metafile Format) formats.

Before opening or saving a PICT file, you must have a FolderItem that represents the PICT file you wish to work with. From there, you can open PICT files with the FolderItem's OpenAsPicture method and save a picture to the file with the SaveAsPicture or SaveAsJPEG methods. If you are working with a vector graphics file, you can use the FolderItem's OpenAsVectorPicture method. REALbasic will try to convert the objects in the file to editable Object2D objects for you.

Saving Pictures

To save a picture to a PICT file, you need a FolderItem that represents a new PICT file or an existing PICT file. Next you call the FolderItem's SaveAsPicture method passing it the picture you wish to save. This example saves the backdrop of a Canvas control to a PICT file, the name of which is specified by the user in a Save As dialog box. The PICT file type is defined in the File Type Set named "ImageTypes".

```
Dim f as FolderItem
f=GetSaveFolderItem(ImageTypes.Pict,"Untitled")
If f <> Nil Then
  f.SaveAsPicture Canvas1.backdrop
End If
```

To save in JPEG format, simply substitute "image/jpeg" as the first parameter of GetSaveFolderItem.

The SaveAsPicture method has an optional second parameter that enables you to specify the format that you want to use for the saved file. This is how you can specify either a vector or bitmap format. For example, you can specify a meta-format that maps to various concrete formats based on the target, the data being saved, or other criteria. The following table gives the codes for meta-formats.

Table 34: Codes, class constants, and descriptions of meta-formats used by the SaveAsPicture method.

Value	Class Constant	Description
0	SaveAsMostCompatible	Most widely-used format for the platform Mac = PICT Win32 = BMP)
1	SaveAsMostComplete	Format most likely to retain all vector info Mac = PICT Win32 = EMF)

Table 34: Codes, class constants, and descriptions of meta-formats used by the SaveAsPicture method. (Continued)

Value	Class Constant	Description
2	SaveAsDefault	DefaultVector or DefaultRaster, depending on picture data Mac = PICT Win32 vector=EMF Win32 raster= BMP
3	SaveAsDefaultVector	Platform's standard vector format Mac = PICT Win32 = EMF)
4	SaveAsDefaultRaster	Platform's standard raster format Mac = Raster PICT Win32 = BMP)

In your code, you can use a class constant in place of the numeric value for the format. Please refer to the FolderItem entry for more information on saving in raster or vector formats.

Saving the image drawn into the graphics property of a Canvas control (perhaps by its Paint event handler) is a bit trickier. That's because the graphics property isn't a picture. The way to solve this is to add a picture property to the window. Any drawing you do in the Canvas control's graphics property should also be drawn into the picture property. The picture can then be saved using the SaveAsPicture method. The picture property you add to the window must be filled with a reference to a new picture before you attempt to write to it. This is accomplished using the NewPicture function in the window's Open event handler. In this example, the picture property (called "p") is set to a new picture:

```
p=newpicture(Canvas1.width,Canvas1.height,32)
```

In this example, the MouseDown event handler of the Canvas1 control draws a black pixel when the user clicks on the Canvas1 control. The drawing is also done to the window's p (picture) property:

```
Me.Graphics.Pixel(x,y)=Rgb(0,0,0)
p.Graphics.Pixel(x,y)=Rgb(0,0,0)
```

Finally, the picture property "p" can be saved to a picture file:

```
Dim f as FolderItem
f=GetSaveFolderItem("image/x-pict","Untitled")
If f <> Nil Then
  f.SaveAsPicture p
End If
```

Opening Pictures

To open a picture, you need a FolderItem that represents the image file you wish to open. If you are working in a cross-platform situation, consider using BMP files

because REALbasic can read this format without QuickTime being installed on the user's computer.

To open the picture, call the FolderItem's OpenAsPicture method which returns the picture. If QuickTime is installed, OpenAsPicture will open any kind of graphics file QuickTime will open (JPEG, GIF, etc.). QuickTime is not required for opening JPEG, GIF, and BMP files on Windows. OpenAsPicture does not open JPEG images on Linux.

This example displays the Open File dialog box that lets the user choose a file which is then placed in the Backdrop property of a Canvas control. The .bmp and .pict file types are specified in the ImageTypes File Type Set.

```
Dim f as FolderItem
f=GetOpenFolderItem(ImageTypes.All)
If f <> Nil Then
  Canvas1.Backdrop=f.OpenAsPicture
End if
```

If the file consists of vector graphics, you can call OpenAsVectorPicture instead of OpenAsPicture to ask REALbasic to try to map the objects in the file into REALbasic vector graphic objects. This option is available for vector PICT files on Macintosh and emf files on Windows.

The original file may have objects for which there is no equivalent in REALbasic. REALbasic will do its best to map these objects, but there may be some loss of information, depending on the characteristics of the original file. PICTs support unrotated Rectangles, Lines, Ellipses, RoundRects, Polygons, Text, Pixmaps, and Arcs.

Extended Metafile files (.emf) support unrotated Rectangles, Lines, Ellipses, RoundRects, Polygons, Text, Pixmaps, and Arcs.

Files of type .emf are displayed as actual size. For the most part this is huge; you probably will want to scale them down before viewing (pic1.objects.scale = scalingFactor). We find that a scale factor of 0.045 is a good value.

Working With Sound Files

REALbasic supports opening Macintosh and WAV sound files but not saving them. Specifically, Macintosh sound files are those files whose "Kind" field in the file's Get Info dialog box is listed as "Sound." To open a sound file, you must first have a FolderItem that represents the sound file you wish to open. Next, you can open the

sound file and place its contents into a Sound object with the FolderItem's OpenAsSound method. This example opens a sound file and plays it:

```
Dim f as FolderItem
Dim s as Sound
f=GetFolderItem("Doh!")
If f<> Nil Then
  s=f.OpenAsSound
  s.Play
End if
```

You can also get sounds stored in a snd resource inside your application. For more information, See "Supported Resource Types" on page 414.

Working With Movie Files

To open a movie file, you must first have a FolderItem that represents the file you wish to open. REALbasic supports the QuickTime player on Macintosh and Windows Media Player on Windows. Next, you can open the file and place its contents into a Movie object with the FolderItem's OpenAsMovie method. This example opens a QuickTime file, assigns its movie to the Movie property of a MoviePlayer control, and plays the movie:

```
Dim f as FolderItem
Dim m as Movie
f=GetOpenFolderItem(VideoTypes.QuickTime)
If f<> Nil Then
  m=f.OpenAsMovie
  moviePlayer1.Movie=m
  moviePlayer1.Play
End if
```

NOTE: If your application needs a specific QuickTime movie, you can drag it into the Project Editor rather than use GetOpenFolderItem or GetFolderItem.

If you want to edit the movie within REALbasic, you need to open it as an EditableMovie. You would then display the movie in a MoviePlayer by assigning it to the Movie property of a MoviePlayer control.

The following example opens an existing movie as an EditableMovie and displays it in a MoviePlayer control named ThePlayer.

```
Dim f As FolderItem
Dim theEMovie as EditableMovie
f=GetOpenFolderItem(VideoTypes.QuickTime)
If f<>Nil and f.exists then
 theEMovie=f.OpenEditableMovie
 If theEMovie<>Nil then
  ThePlayer.movie=theEMovie
 end if
end if
```

The EditableMovie class includes methods that allow you to:

- Define a segment in the EditableMovie,
- Cut or copy the segment to the Clipboard,
- Paste a segment into the current movie,
- Append another movie segment to the current movie,
- Insert a segment into the current EditableMovie at a specified position,
- Create new sound and/or video tracks,
- Scale the video track.

You can save your changes to the EditableMovie automatically when you are finished or at any time by calling the CommitChanges method. Please see the EditableMovie entry in the Language Reference for more information on the methods used to work with QuickTime movies.

Working With Binary Files

Binary files are simply files that store values in their binary format rather than as text. For example, the number 30000 stored as text requires 5 characters of text (or bytes) to store in a text file. In a binary file, this number can be written as a short integer (or just "short"). A short requires only 2 bytes.

Binary files also have the added benefit that you can read and write to a file without having to close the file in-between. For example, you can open a binary file, read some data, then write some data, and close it. You can also read and write anywhere in the file without having to read through all the data preceding the data you want.

Most applications store data in a binary format. The format is simply the arrangement of data within the file. In order to read a binary file, you must know how the data is arranged. If your own application created the file, you will know this, but if the file was created by an application you didn't write, you may not know it. Some formats are made public. For example, the PICT format is public.

Other formats are not. Many software vendors do not publish the binary formats that their applications use to create documents.

BinaryStreams

Data read from or written to a binary file travels through a *BinaryStream*. A BinaryStream is a class of object in REALbasic that represents the flow of information between the FolderItem and the file it represents. Unlike the TextInputStream class (which can only be used to read from a text file) and the TextOutputStream class (which can only be used to write data to a text file), BinaryStreams can be used for both reading data and writing data. You can even indicate to the BinaryStream that you will only be reading data from the file so that the file can continue to be available to other applications for writing.

BinaryStreams can read and write specific types of data, such as strings, short integers, long integers, and single bytes. They can also be used to read and write raw unformatted binary data.

Reading From a Binary File

Once you have a FolderItem that represents the file you wish to open, you open the file using the OpenAsBinaryFile method of the FolderItem class. This method is a function that returns a BinaryStream. You then use Read, ReadByte, ReadLong, ReadPString and ReadShort methods to read data from the stream. The BinaryStream keeps track of the last position in the file you read from in its Position property. However, you can change this property's value to move the position to any location in the file.

This example presents the Open File dialog box, reads a file made up of strings, and displays those strings in a ListBox. Notice that since the code is only reading data and not writing, False is passed to the OpenAsBinaryFile method to indicate the file should be opened in "read-only" mode. Also, reading continues in a loop until the stream's EOF (end of file) property is True. REALbasic will set the EOF property to True automatically once the end of the file is reached.

```
Dim f as FolderItem
Dim stream as BinaryStream
f=GetOpenFolderItem("myFileType")
If f<> Nil Then
  ListBox1.DeleteAllRows
  stream=f.OpenAsBinaryFile(False)
  do
    ListBox1.AddRow stream.ReadPString
    ListBox1.Cell(ListBox1.ListCount-1,1)= stream.ReadPString
  Loop Until stream.EOF
  stream.Close
End if
```

This code would run about 25% faster using a For...Next loop instead of a Do loop. However, the format of the file would have to be different because you need to know in advance how many rows of data to read in order to provide the ending value to

the For loop. For example, if the first four bytes of the file format was a long integer that was the number of rows in the file, you could use that integer in your For loop. This is illustrated in the following example:

```
Dim f as FolderItem
Dim stream as BinaryStream
Dim count,i as Integer
f=GetOpenFolderItem("myFileType")
If f<> Nil Then
  ListBox1.DeleteAllRows
  stream=f.OpenAsBinaryFile(False)
  count=stream.ReadLong
  For i=1 to count
    ListBox1.AddRow stream.ReadPString
    ListBox1.Cell(ListBox1.Listcount-1,1)=stream.ReadPString
  Next
  stream.Close
End if
```

When you read a BinaryStream, you may need to take the encoding of the characters into account. To do so, you can pass an optional parameter to the Read and ReadPString methods that specifies the encoding. Use the Encodings object to get any encoding and pass it to Read or ReadPString. For more information, see the section "Working with Text Encodings" on page 333.

Writing to a Binary File

Once you have a FolderItem that represents the file you wish to open and write to, you can open the file using the OpenAsBinaryFile method of the FolderItem class. If you are creating a new file, use the CreateBinaryFile method of the FolderItem. This method is a function that returns a BinaryStream. You then use Write, WriteByte, WriteLong, WritePString, and WriteShort methods to write data to the stream. The BinaryStream keeps track of the last position in the file you wrote to in its Position property. However, you can change this property's value to move the position to any location in the file.

When you are finished writing data to the file, call the BinaryStream's Close method to close the stream to the file making the file available to be opened again.

This example displays the Save As dialog box and writes the contents of two columns of a ListBox to the file and closes the stream. This code creates the file that is opened and read in the read binary file example that uses a For...Next loop.

```
Dim f as FolderItem
Dim i as Integer
Dim stream as BinaryStream
f=GetSaveFolderItem("myFileType","Untitled")
If f<> Nil Then
  stream=f.CreateBinaryFile("myFileType")
  stream.WriteLong ListBox1.ListCount
  For i=0 to ListBox1.Listcount-1
    stream.WritePString ListBox1.List(i)
    stream.WritePString ListBox1.Cell(i,1)
  Next
  stream.Close
End if
```

Working With Virtual Volumes

Up to this point, this chapter has discussed standard desktop files and folders that can be created, read from, and written to by any application that is designed to operate on the specific file type. REALbasic also allows you to create and maintain a special type of document that is capable of holding several different files. In this sense, it is more like a volume than an individual file. For that reason, they are called *virtual volumes*. A virtual volume can be created, written to, and read from by a REALbasic application. Virtual volumes are fully supported across all platforms, so you can create a virtual volume on a Macintosh platform and read from it, modify existing files, or write additional files to it on a Windows platform.

The VirtualVolume class enables you to create and maintain a hierarchy of "virtual" files within one physical file. Basic reading and writing text and binary streams are supported. That is, you can create either text or binary files using the TextOutputStream and BinaryStream classes. This means that the SaveAsPicture and SaveStyledEditField methods of the FolderItem class do not work for virtual volumes.

The FolderItem class has two methods and one property for working with virtual volumes. Use the CreateAsVirtualVolume and OpenAsVirtualVolume methods to create and open virtual volumes. The VirtualVolume property of the FolderItem class returns the VirtualVolume if the FolderItem is in a virtual volume.

Once you've created a virtual volume, you can get its Root property and navigate to it using the same FolderItem methods that you would for real files. Virtual volumes do not support resource forks.

The following code creates a VirtualVolume and gets its Root property. Once you have the virtual volume's root, you can write files relative to that root using the TextOutputStream or BinaryStream classes, as described earlier in this chapter.

```
Dim f as FolderItem
Dim v as VirtualVolume
v=New VirtualVolume
f=GetFolderItem("").Child("VV")
If f <> Nil then
 v=f.CreateVirtualVolume
 if v.Root <> Nil then
  MsgBox v.Root.Name
 End if
End if
```

Working With Macintosh Resources

All Macintosh files (including applications, which are really just files) can have two sections called "forks." The "data" fork holds data that is in whatever format the application that created the file chose to put it in. The resource fork can contain formatted information such as icons, sounds, menu bars, pictures, string lists, etc.

REALbasic provides support for reading from and writing to the resource fork of a file. This is done using a FolderItem class object that represents the file whose resource fork you wish to access or create.

If you need more information on Macintosh resources, read *Inside Macintosh: Resources* published by Addison-Wesley.

REALbasic supports operations on the resource fork only on the Macintosh platform. The discussion and examples in this section assume that the application is running on a Macintosh. If your application manages resources, you can use the TargetWin32 or TargetMacOS constants to verify that the application is running on Macintosh before doing any resource fork operations. There is one exception to this: it is possible to install custom cursors in your application using the 'CURS' resource and access them from a built Windows application. This is described in the section "Custom Cursors in Windows Applications" on page 415.

Opening a File's Resource Fork

Once you have a FolderItem, you can open the resource fork for the file the FolderItem represents. This is done using the OpenResourceFork method of the FolderItem. This method returns a ResourceFork class object which can then be used to access the resource fork of the file. If the file has no resource fork, the OpenResourceFork method returns Nil.

This example displays the Open File dialog box, allowing the user to choose a file. It then reports if the file has no resource fork or tells the user how many different types of resources are in the file's resource fork:

```
Dim f as FolderItem
Dim rf as ResourceFork
f=GetOpenFolderItem("any")
If f <> Nil Then
  rf=f.OpenResourceFork
  If rf=Nil Then
    Beep
    MsgBox "This file has no resource fork."
  Else
    MsgBox "This file has "+Str(rf.TypeCount)+" resource types."
  End if
End if
```

NOTE: The "any" file type passed to GetOpenFolderItem was defined in the File Type Sets Editor or with the FileType class via the language and uses the string "????" as both Type and Creator. The "?" is a wildcard character, matching any type or creator code.

Adding a Resource Fork to a File

Before you can write to the resource fork of a file, it must have one first. You can use the FolderItem's OpenResourceFork method to determine if the file has a resource fork. If it doesn't, you can use the FolderItem's CreateResourceFork method to add a resource fork to the FolderItem. Once the file has a resource fork, you can begin writing to it.

This example displays an Open File dialog box and adds a resource fork to the file (if the file doesn't already have one):

```
Dim f as FolderItem
Dim rf as ResourceFork
f=GetOpenFolderItem("any")
If f <> Nil Then
  rf=f.OpenResourceFork
  If rf=Nil Then
    rf=f.CreateResourceFork("any")
  End if
End if
```

Adding a Resource Fork to a Project

You can add a resource fork to a REALbasic project by dragging a resource file into the Project Editor. REALbasic recognizes the file type of "rsrc" as a resources file. You can have as many resource files in your project as you wish. The resources from all your resource files will be copied into the built Macintosh application. In the case of a conflict, later resource files overwrite earlier ones, where the files are written in the order in which they appear in the Project Editor.

The following example opens the application's resource fork:

```
Dim rf as ResourceFork
rf=App.ResourceFork
```

Supported Resource Types

REALbasic provides high level support for PICT, CICN, CURS, and snd resources. You can use the AddPicture method to add PICT resources to the resource fork and use GetPicture or GetNamedPicture to get PICT resources from the resource fork. You can use GetCicn to get a cicn (color icon) resource. Sounds can be read from snd resources using the GetSound method of the ResourceFork class. However, you can access any type of resource. REALbasic provides method for getting and setting raw data from any type of resource in a resource fork. However, you must know the format of the resource data to be able to successfully read from it or write to it.

Reading Resources

The ResourceFork class has methods for reading data from four different types of resources. You can read PICT resources using the GetPicture and GetNamedPicture methods of the ResourceFork class. You can get a color icon as a picture by calling the GetCicn method. You can get a large (32 x 32) icon using the GetIcl method and a small (16 x 16) icon with the GetIcs method. For example, the following line of code displays a picture resource in an ImageWell:

```
Me.Image=App.ResourceFork.GetPicture(128)
```

Use GetCicn, GetIcl, and GetIcs in the same way.
You can load sounds from 'snd ' resources using the GetSound method of the ResourceFork class.

Reading Custom Cursors

Use GetCursor to assign a custom cursor to the MouseCursor property of the Application, a Window, or a Control. For example, the following line in a window's MouseEnter event handler assigns a custom cursor to the MouseCursor property of the window.

```
self.MouseCursor=App.ResourceFork.GetCursor(128)
```

This line causes REALbasic to display the custom cursor whenever the mouse enters the window's region. If you want to change the cursor when the mouse is over a control in the window, you can use a line such as this in the control's MouseEnter event handler:

```
self.MouseCursor=App.ResourceFork.GetCursor(129)
```

and then restore the window's custom cursor with the following line in the control's MouseExit event handler:

```
self.MouseCursor=App.ResourceFork.GetCursor(128)
```

Notice that these lines do not assign a cursor to the control's own MouseCursor property; they only change the assignment to the control's parent window.

Assign a custom cursor to a control's MouseCursor property only when you don't use either the Window's or the App class's MouseCursor properties. If either the parent window or the application has a custom MouseCursor property, then the control's MouseCursor property is ignored.

To assign a custom cursor to the application as a whole, create a new class called "App" and make its Super Class "Application." Then add a line such as this to the App class's Open event handler.

```
MouseCursor=App.ResourceFork.GetCursor(128)
```

This causes the REALbasic application to display this custom cursor all the time. That is, the MouseCursor properties of any of the application's windows or controls will be ignored.

Custom Cursors in Windows Applications

You can use CURS resources to assign custom cursors in built Windows applications. This, at present, is the only case in which resources are supported on Windows builds. The relationships among the MouseCursor properties of the Application, Window, and Control are the same as for Macintosh applications, but you must create the resource files in a special way. This technique is also recommended for Macintosh builds.

The key is that you must create a separate resource file for each custom cursor that you add to the project. Each resource file must have only a CURS resource that contains one cursor. Typically, you will assign the custom cursor to ID 128.

You then drag all of these resource files to your project. These special resource files will appear in the Project Editor with a cursor icon.

You can then access the custom cursors by referencing their names. For example, the following statement in a control's MouseEnter event handler changes the pointer to the Sponge cursor when the mouse enters the region of the control:

```
Self.MouseCursor=Sponge
```

(You would then restore the cursor with a statement in the control's MouseExit event handler.)

Reading Other Resources

To read data from other resources, you must know the format of the resource. For example, to read the STR# resource, you can use the GetResource method of a ResourceFork class. This will return the bytes that make up the resource ID you specify. To then do anything useful with the data, you will need to know that the first two bytes are the number of strings in the resource, followed by the strings themselves. The strings are Pascal strings so their first byte is the length of the string.

When you build your application, you can enter version information about the application in the Build Application dialog box. That information is stored in a 'vers' resource that becomes part of your application. An application can access the 'vers' resource using the GetResource method of the ResourceFork class. The information written to the 'vers' resource is described in the section "Version Information" on page 553.

Writing To Resources

REALbasic provides methods via the ResourceFork class that can be used to write to resources. You can use the AddPicture method to write REALbasic pictures into a PICT resource. For all other types of resources, you can use the AddResource method to create new resources and the RemoveResource method to delete specific resources. To modify a resource other than PICT resources, you read the data of the resource using the GetResource method, then write the data back by deleting the resource with the RemoveResource method and then recreating the resource using the AddResource method.

More Information on the ResourceFork

For more information on the ResourceFork class, see the GetResourceFork class in the *Language Reference*.

Files Opened From the Desktop

If your application is designed to read from and/or write to files, you may need to consider how your application will react when the user accesses files stored on his/her computer.

Files Opened by Double-Clicking

If the user double-clicks on a file whose creator code matches your stand-alone application's creator code, the user will be expecting your application to open the file automatically. If your application is prepared to open a file and take some action, then you should also support the user's double-clicking on the file from the desktop. This is done by adding a new class based on the Application class. This new class represents your application as a whole and will receive information when the user double-clicks on a document whose creator code matches your application's creator code. The application class you are adding has an OpenDocument event handler that is executed when the user double-clicks on a file at the desktop. This event handler is passed a FolderItem as a parameter. This FolderItem represents the file the user double-clicked on.

To take action when the user double-clicks on a file from the desktop, do this:

1 **Double-click the App object in the Project Editor.**

By default, a project created from the default project template has an App class based on the Application class. If you don't have an App class, create a new class and set its Super Class to Application in its Properties pane.

2 **Expand the Events list in the Code Editor browser.**

3 **Click on the OpenDocument event to select it.**

4 **Enter the code that should execute when the user double-clicks on a file at the desktop. You can access the file using the item parameter passed to the OpenDocument event handler.**

Files Dropped On Your Application's Icon

REALbasic treats a file dropped on your application's icon at the desktop the same way it treats the user's double-clicking on a file from the desktop. For more information, See "Files Opened by Double-Clicking" on page 416.

Creating New Files

When the user launches your application without opening a file, REALbasic assumes that the user will probably want to create a document (assuming you application is document/file based). If you have created a class based on the Application class, that class's NewDocument event handler will execute. On Macintosh, this event handler also executes when your application receives an Open Application AppleEvent (oapp) or when a user uses AppleScript to tell the Finder to open your application.

You can call the NewDocument event handler by entering NewDocument in your code. This allows you to have a single location to put the code for your application that creates new documents. Using this event handler, your application will respond to all the appropriate calls to create a new document.

CHAPTER 9

Creating Reusable Objects with Classes

Classes act as templates for objects much in the same way that the windows listed in the Project Editor act as templates for the windows you open in your application. This chapter will introduce you to the benefits of classes, explain how to modify them, and how you can create custom interface controls using classes.

Contents

The Benefits of Classes

Classes offer lots of benefits. They are:

Reusable Code

When you add code to a PushButton control to customize its behavior, you can only use that code with that PushButton. If you want to use the same code with another PushButton, you need to copy the code and then make changes to the code in case it refers to the original PushButton (since the new PushButton will have a different name than the original).

Classes store the code once and refer to the object (like the PushButton) generically so that the same code can be reused any number of times without modification. If you create a class based on the PushButton control and then add your code to that class, any instances of that custom class will have that code.

Smaller Projects and Applications

Because classes allow you to store code once and use it over and over in a project, your project and the resulting application is smaller in size and may require less memory.

Easier Code Maintenance

Less code means less maintenance. If you have basically the same code in several places in your application, you have to keep that in mind when you make changes or fix bugs. By storing one copy of the code, you will spend less time tracking down all those places in your project where you are using the same code. Making a change to the code in a class automatically updates any places where the class is used.

Easier Debugging

The less code you have, the less code there is to debug.

More Control

Classes give you more control than you can get by adding code to the event handlers of a control in a window. In fact, some classes can even manage menus. You can also use classes to create custom controls. And with classes, you have the option to create versions that don't allow access to the source code of the class, allowing you to create classes you can share or sell to other REALbasic users.

As you can see, there are many benefits to creating classes. Overall, classes make your programming effort more efficient.

Understanding Instances

REALbasic has many classes built-in to it. PushButton, StaticText, EditField, and ListBox are examples of some of the built-in control classes. Control classes are templates for objects that you use in your application's interface. As templates, the classes are abstract in the sense that you do not use any of the templates themselves in applications. Instead, each template serves as an inexhaustible supply of instances of classes. You use these instances.

For example, when you drag an EditField from the Controls list to a window, you create a usable *instance* of that class. The new instance has all the properties and methods that were built into the EditField template. You get all that for free—styled text, multiple lines, scroll bars, and all the rest of it. You customize the particular instance of the EditField by modifying the values of the instance's properties.

Understanding Subclasses

You may find situations where you would like to have an object that is a slightly altered version of one of the built-in classes. For example, you might want a version of the EditField control that disables the Cut and Copy items on the Edit menu, preventing the user from putting sensitive data on the Clipboard. You might want to create a ListBox that, by default, has the months of the year in it. You can create your own versions of these built-in classes by creating *subclasses* that you add to your Project Editor.

There's an important difference between adding an instance of EditField to a window versus adding a subclass based on EditField to your project. In the latter case, you can customize the subclass based on EditField itself and use instances of it in several places in the application. You can also save the customized template so that you can reuse it in other projects.

What is a Subclass?

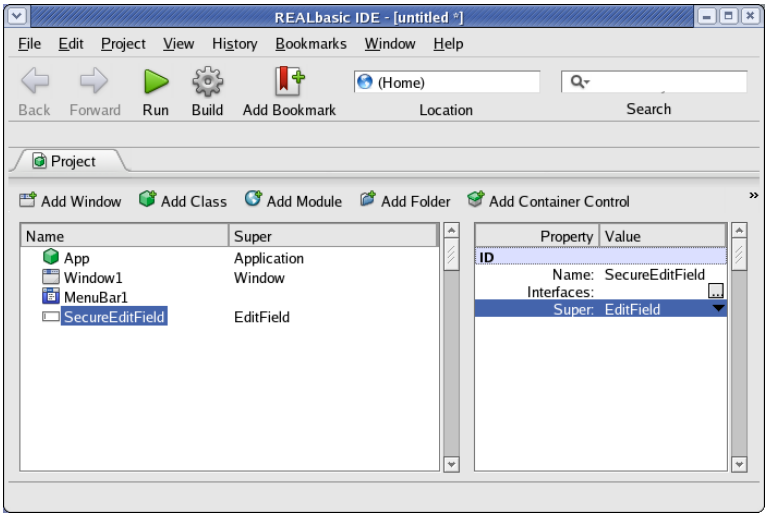
A subclass is simply a class that has a super class. A super class is a class the subclass is based on. The super class is also sometimes called the "parent" class. Subclasses inherit all of their super's properties, methods, constants, and events[1]. The subclass can then modify them. In fact, a subclass is identical to its super class until you start modifying it. After that, it's different from its super class only in the ways you make it different by adding properties, modifying events, and adding or modifying methods.

Examples of Subclasses

For example, to create an EditField that prevents the user from copying data to the Clipboard (Let's call it a SecureEditField), you create a new class and choose EditField as its super class. The new subclass is shown in Figure 295.

1. There is an exception to this rule. If you set the Scope of a method, property, or constant to Private and then use this class as a super class, its subclasses will not inherit the Private methods, properties, or constants.

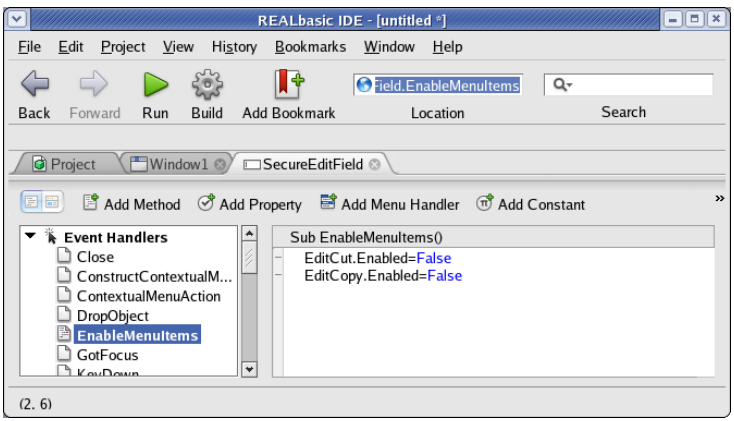
Figure 295. SecureEditField based on the EditField class.



REALbasic automatically enables the Cut and Copy menu items on the Edit menu when characters are selected in an EditField. Since SecureEditField is based on EditField, it inherits these features. Because EditFields can get the focus, any subclass of the EditField control has an EnableMenuItems event handler. This allows SecureEditField to control the menus when it has the focus. To prevent the user from using the Cut and Copy menu items, you set the Enabled property of these menu items to False in your SecureEditField's EnableMenuItems event handler.

In Figure 296, the user has opened the Code Editor for the SecureEditField class and added code to disable the Cut and Copy menu items.

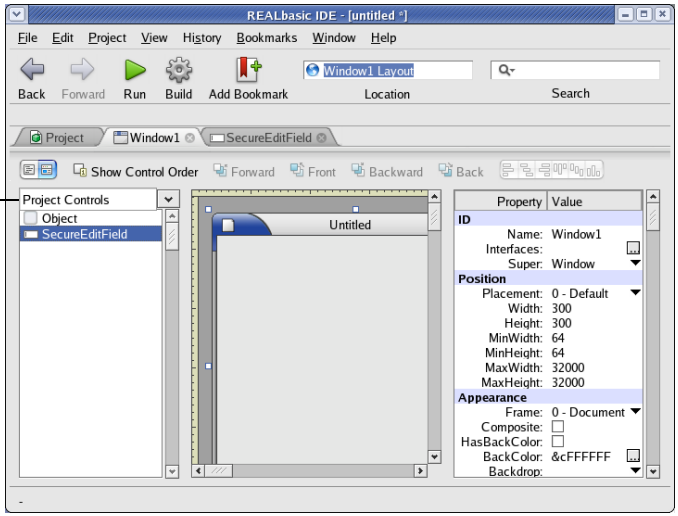
Figure 296. Disabling the Copy and Cut menu items in SecureEditField.



To use an instance of SecureEditField, switch to the Window Editor and use the drop-down menu above the Controls list to display the Project controls. This is the list of custom controls that have been added to the Project Editor.

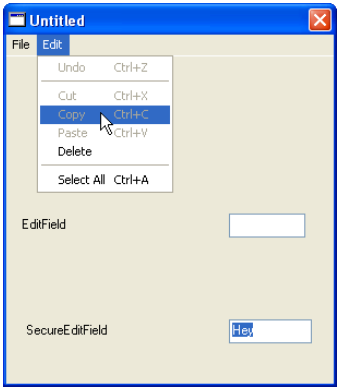
Figure 297. The Project Controls list.

Project Controls is selected from the drop-down list.



Add the SecureEditField item to the window, just as you would a built-in control. It behaves exactly like a 'regular' EditField, except that the Cut and Copy menu items are disabled when the user has selected text.

Figure 298. Selected text cannot be copied from an instance of SecureEditField.

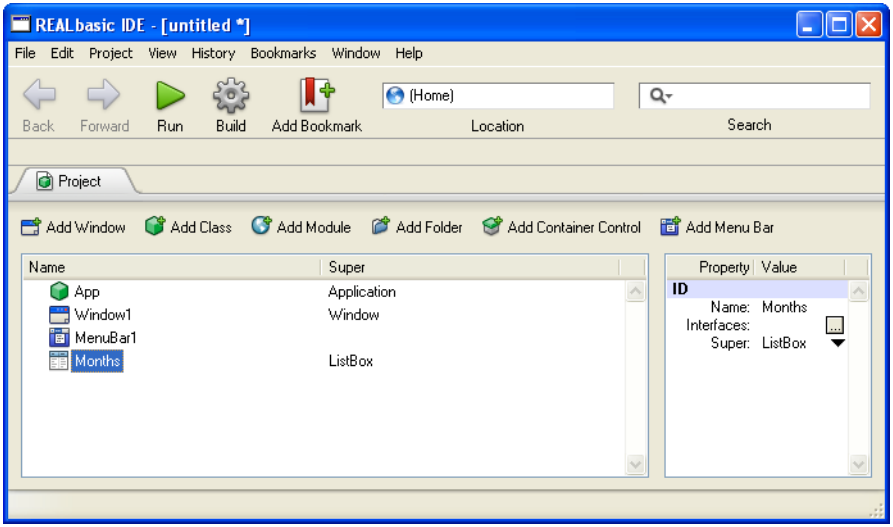


Since SecureEditField is listed in your Project Editor, you can create instances of it anywhere you like and maintain its code in one central place. You can export it and import it into other projects.

Here is another example. Suppose you want to create a ListBox that, by default, displays the names of the months of the year, with the current month selected.

You create a new class and choose ListBox as its super class.

Figure 299. The Months class added to the Project Editor.



Open the Code Editor for the Months class. In the Open event handler of your new subclass add the month names, the heading, and code that selects the appropriate month in the list.

In this case, the Open event handler is:

```
Me.HasHeading=True
Me.InitialValue="Months"

Me.AddRow "January"
Me.AddRow "February"
Me.Addrow "March"
Me.Addrow "April"
Me.Addrow "May"
Me.Addrow "June"
Me.Addrow "July"
Me.Addrow "August"
Me.Addrow "September"
Me.Addrow "October"
Me.Addrow "November"
Me.Addrow "December"

Dim d as New Date
Me.Selected(d.Month-1)=True
```

To add an instance of the Months class to a window, switch to the window's Window Editor and then use the drop-down list above the Controls list to switch to Project Controls. The Months class will be listed there. Drag it to the window. When you run the application, the Open event handler will run and populate the

Listbox with the months of the year. The last two lines of the Open event handler will highlight the current month.

You might want to create an EditField that only allows the user to enter numbers. Let's call it "NumbersOnlyEditField." To do this, you create a subclass of the EditField control and put code in the KeyDown event handler that allows only numbers and rejects all other characters. Once created, you can use your new subclass in many different places in your project, but the code exists only in one place.

Once created, custom control classes can be exported as self-contained objects that can be used in other projects. Just right+click (Control-click on Macintosh) on the custom control class in the Project Editor and choose Export... from the contextual menu.

Subclasses are classes. They are called subclasses to differentiate them from built-in classes and emphasize the point that they inherit the properties, events, and methods of their parent class. Because subclasses are classes, they can be the super class to other subclasses. For example, suppose you had already created the NumbersOnlyEditField subclass mentioned earlier. Now, you need an EditField that allows only numbers within a certain range. You could duplicate the NumbersOnlyEditField subclass and then modify its code. However, this would make your project larger and more difficult to maintain. If you found a bug in the code of the NumbersOnlyEditField, you would have to remember that you used that code in other places as well, track them down, and fix them. A more efficient way is to create a new subclass and choose the NumbersOnlyEditField as its super class. The new subclass (let's call it "NumberRangeEditField") would utilize all of the properties, events, and methods of its super class. However, you can add code to the TextChanged event handler that allows only numbers within a specific range.

Referring to a Class's Properties and Methods From Within the Class

When you add control such as a PushButton to a window and then add code to that control, you are really adding code to one instance of the PushButton class. Consequently, you must include some reference to the instance or REALbasic would have no way of knowing which PushButton, for example, your code is referring to. You use the Name property of the instance to do this.

For example, when you add the an instance of a PushButton control to a window, REALbasic gives the instance the default name of "PushButton1". Suppose you add a PushButton named "PushButton1" to a window that should be disabled after the user clicks it. The code in the PushButton's Action event handler would be:

```
PushButton1.Enabled=False
```

However, when you add code to a class or subclass, there is no need to refer to any instance. For example, if you add a custom class to the Project and set its Super Class to PushButton, you are creating a customized PushButton that you can use in many

places in your application. All of the code that you add to the custom class will automatically become part of all instances of that custom class.

Consequently, you don't include object references to the custom class in its own code. If you had added a custom class to the Project with PushButton as its Super class, you would not include an instance reference. The code would be:

```
Enabled=False
```

When you add an instance of that custom PushButton to a window, the code added to the class will automatically be operating on the instance of the class. In this example, the customized PushButton is the same as a 'regular' PushButton except that it has its own Action event handler. The code you added to the class becomes the Action event handler for all instances of the custom PushButton class.

Creating Classes

Adding a class to a project is easy. If you want to subclass the class from an existing class in the project, use the steps in the section that follows immediately.

To add a new class, do this:

1. **If it is not already visible, click on the Project tab to display the Project Editor.**
2. **Click the Add Class button in the Project Editor toolbar or choose Project ▸ Add ▸ Class.**

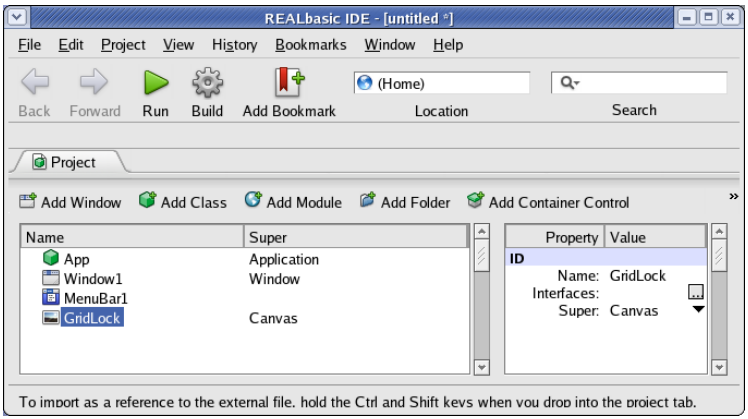
   A new class, named Class1, is added to the Project Editor. When the new class is selected, the Properties pane changes to indicate the new class's properties.

   By default, the new class is not subclassed from any other class. Often you want the new class to inherit the properties of an existing class.
3. **Use the Properties pane to give the new class a meaningful name and, if desired, use the Super pop-up menu to subclass it from an existing class.**

   If the Super class is one of the built-in controls, the new subclass gets the icon belonging to that control in the Project Editor.

   The "Gridlock" example on the CD uses a custom control to draw a rectangular grid in a window. In Figure 300 on page 427 we see that this control is based on the Canvas control.

Figure 300. The Gridlock class is based on the Canvas control.



With the new class selected in the Project Editor, you can double-click it to open the Code Editor for the new class.

Creating a Subclass from an Existing Class

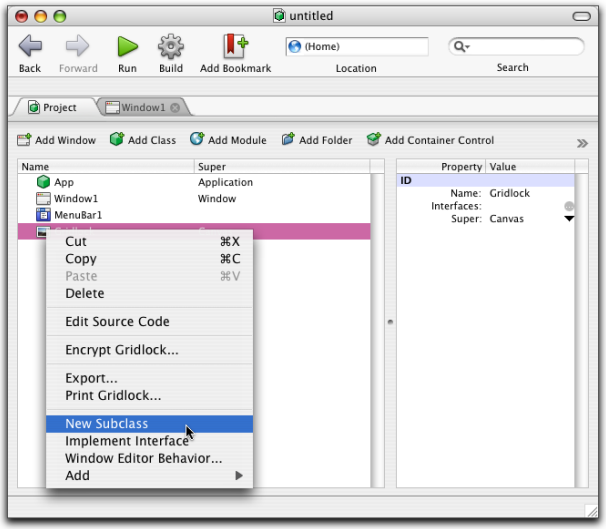
When you want to create a subclass of an existing class that you've already added to the Project Editor, you can do so using these two shortcuts.

To create a subclass of an existing class in the Project Editor, do this:

1. **Right+click (Control-click on Macintosh) on the class from which you want to create a subclass.**
2. **Choose New Subclass from the contextual menu.**

   A new class is added to the project. In the Properties pane, the existing class is used as the Super class of the new class.

Figure 301. Using the Project Editor contextual menu to create a new class.



3 Use the Properties pane to rename the new subclass.

You can also create a subclass of an existing class by repeating the procedure for creating a new class and then manually choosing the parent class from the Super pop-up menu in the Properties pane.

To create a subclass of a control class, do this:

1 Drag the control from the Controls list to the Project panel.

A new class derived from the control is added to the project.

2 Use the Properties panel to rename the class.

This method could be used to add the GridLock class to the project in Figure 300.

Saving Classes

Since classes are reusable, you will want to develop a library of classes that you can easily import into other projects. On Windows and Linux, choose File ▶ Export and save the class under a suitable name. On Macintosh, you have the option of dragging the item from the Project Editor to the Finder. To add the class to another project, simply drag it from the desktop into the Project Editor for the new project or choose File ▶ Import and use the open-file dialog box to choose the class to be imported. For more information, see "Exporting Classes For Use In Other Projects" on page 472.

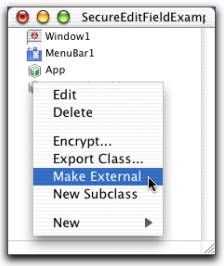
External Project Items

If you want to use the class in more than one project, you can export it as an external project item. An external project item is stored on disk and is referenced by each project that uses it. If a change to the class is made from one project, those changes

are made available to all the other projects that reference the external item. Changes to the external project item are saved to disk when you save the project.

To save a class in your project as an external project item, Right+click (Windows and Linux) or Control-click (Macintosh) on the class in the Project panel and choose the Make External contextual menu item. An Export File dialog box appears in which you can save the class to disk. When you complete the save, the class appears in your Project panel with an alias badge and the name of the class is in italics. This indicates that it is now referenced as an external item.

Figure 302. The SecureEditField class as an external project item.



Choose Make External from the contextual menu…

After you save the class as an external project item, it appears as an alias.

To add an external project item to another project, hold down the Ctrl+Shift on Windows and Linux or (⌘ and Option keys on Macintosh while you drag the item from the desktop to the Project panel. In the Project panel, the external project item's name will be shown in italics.

For more information, see the section, "External Project Items" on page 56.

Modifying Classes

One of the big advantages of classes is the ability to modify existing classes. You do this by adding constants, properties, adding or changing events, and adding or changing methods.

Scope of a Class's Methods, Properties, and Constants

When you add a method, property, or constant to a class, you need to set its Scope attribute. The Scope of a method, property, or constant determines which other items in the project can access it. There are three possible values:

- **Public: Accessible from Anywhere** A Public method, property, or constant is available to code throughout the application. When you need to access a Public item, you use the "dot" notation. For example, if you declare a Public property, myPublicProperty, in Class1, you call it by referring to "Class1.myPublicProperty"

outside Class1. Within a method or event handler belonging to Class1, it is in scope, so you can access it by referring to "myPublicProperty".

- **Protected: Accessible from the Current Class and its Subclasses**: A Protected method, property, or constant is available only to other code within the class and subclasses based on this class. It is "invisible" to the rest of the application. When other code in the class needs to access a Protected method, property, or constant, you simply reference it by name. If you try to access a Protected method, property, or constant outside of the class, REALbasic will display an informative error message indicating that the item is out of scope.
- **Private: Accessible from the Current Class only**: A Private method, property, or constant is like a Protected item except it is not accessible to classes subclassed from the current class. Public and Protected classes are accessible to classes subclassed from the current class.

Adding Properties

You can add properties to a class to store values that its super class doesn't store. For example, you might want to create a subclass of the EditField control that stores the last value the user entered. This would allow you to selectively reject the current entry and restore the last entry. You add properties to a class the same way you add properties to a window.

To add a property to a class, do this:

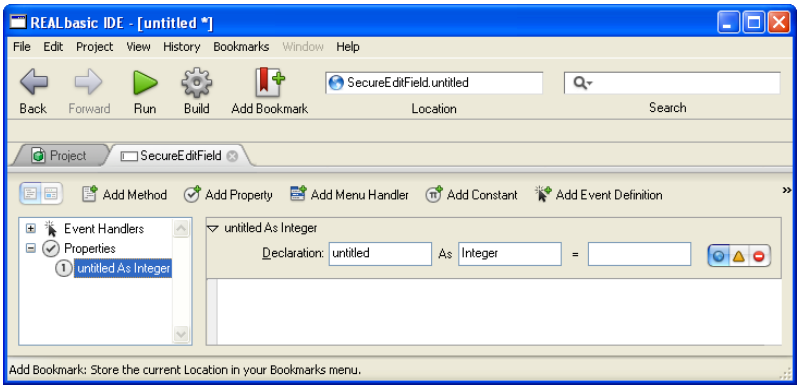
1. **If the class's Code Editor is not already open, double-click on the class in the Project Editor to open it.**

   The Code Editor for the class appears.

2. **Click the Add Property button or choose Project ▸ Add ▸ Property.**

   The Property declaration area appears above the Code Editor area. A "placeholder" property declaration is entered by default.

Figure 303. The Property Declaration area.



The Property Declaration area has three fields. They are for the name of the property, its data type, and its default value. The first two are required. If you do

not provide a default value, the new property will take the default value for the data type that you choose. Strings have a default value of a empty string, numbers have a default value of zero, booleans have a default value of false, colors have a default value of black, and objects have a default value of Nil.

3 Fill in the Name and Data Type fields and, if desired, provide a default value.

The property can be an array. For example, if you want to declare a four-element String array of first and last names, addresses, and phone numbers called aNames, you would write:

```
aNames(3) as String
```

in the Declaration area. You can declare an array with no elements by using empty parentheses. You code can modify the number of elements. For example,

```
aNames() as String
```

4 Choose a Scope for the property by clicking one of the three Scope buttons.

Your choices, from left to right, are Public, Protected, and Private.

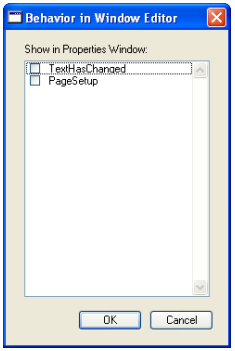
Figure 304. The Scope buttons.

For information about Scope, see the section, "Scope of a Class's Methods, Properties, and Constants" on page 429.

5 (Optional) If you want the property to be listed in the Properties pane in the Window Editor, right+click (Control-click on Macintosh) the name of any property in the browser and choose Window Editor Behavior.

The Behavior in Window Editor dialog box appears. it lists all the properties that you have declared for the class that can be set in the IDE. Some properties can only be set in code.

Figure 305. The Behavior in Window Editor dialog box.



6 Check each property that you want to have listed in the Properties pane.

Selecting the "Show in Properties Window" option means that, if you drag an instance of the class onto a window, the property appears as an item in the Properties pane and can be assigned a value from there, rather than only with code.

7 (Optional) In the Code Editor area, add notes and comments about the property.

The text entered into the Code Editor for a property is automatically non-executable, even if you write valid REALbasic code. Add any comments you wish, including code samples. For an example, see the section "Documenting Properties" on page 193.

Adding Computed Properties

A computed property is actually made up of a pair of methods called Get and Set. It does not store a value in the instance; it does the calculations you program. Obviously this blurs the distinction between properties and methods. The Set method sets the value of a property (writes) and Get reads a value. You can implement either or both, making the computed property Read Only, Write Only, or Read/Write.

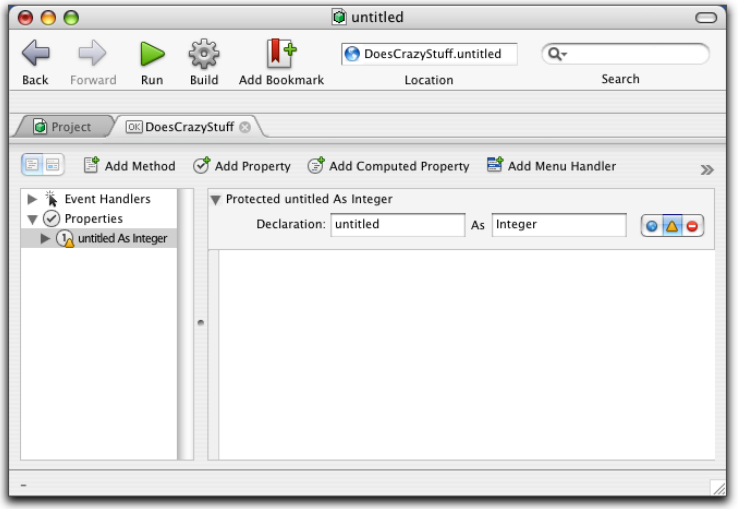
Computed properties can be declared as shared computed properties. See the following section, "Adding Shared Properties" on page 434 for information on shared properties.

To create a computed property, do this:

1 Click the Add Computed Property button in the Code Editor toolbar or choose Project ▶ Add ▶ Computed Property.

REALbasic adds an untitled property to the window and displays a Property Declaration area above the Code Editor.

Figure 306. The Declaration Area for a Computed Property.



2 **Enter the Name and Data Type for the computed property and set its scope.**

Notice that the browser area shows that you can expand the computed property.

3 **Click the plus sign (Windows) or the disclosure triangle (Macintosh and Linux) to reveal the Get and Set methods that belong to the computed property.**

The Get method returns a value of the declared data type. The Set method is passed the value of the property. You do not have to implement both methods. You can make the computed property Read Only, Write Only, or Read/Write.

4 **Write either the Get or the Set method or implement both methods.**

An Example Computed Property

Here is a simple example of a computed property in a custom control class. It implements an annoying PushButton control that disables itself whenever the mouse pointer enters its region and enables itself when the mouse pointer exits its region. This makes it impossible to click, but it appears to be enabled until you try to use it. Don't try this at home.

First, add a class to the Project Editor and set its Super class to PushButton. Double-click it to open its Code Editor. Add a property to the class called mDoesTheCrazies and give it a data type of Boolean. Then add a Computed Property to the class called DoesCrazyStuff and also give it a data type of Boolean.

Expand DoesCrazyStuff in the Code Editor browser and enter the following line into its Get method:

```
Return mDoesTheCrazies
```

It gets the current value in the 'regular' property, mDoesTheCrazies.

The Set method passes the current value in as the parameter, *Value*. Enter the following line as the Set method:

```
mDoesTheCrazies = Value
```

In the MouseEnter event for the custom PushButton class, add the code:

```
If DoesCrazyStuff then me.Enabled=False
```

In the MouseExit event, enter the line:

```
If DoesCrazyStuff then me.Enabled=True
```

In other words, the "regular" property, mDoesTheCrazies, stores the boolean value that is set by the computed property. The Get method gets the current value of mDoesTheCrazies. The MouseEnter and MouseExit methods control the Enabled property of the custom PushButton depending on the computed property.

Adding Shared Properties

A shared property is like a 'regular' property, except it belongs to the class, not an instance of the class. A shared property can be read or set from any instance of the class or from the class itself.

In contrast, "regular" properties are considered *instance* properties. This means that they belong to a particular instance of the class.

The key advantage of shared properties is that you can share them among all the instances of the class. For example, if you are using an instance of a class to keep track of items (e.g., persons, merchandise, sales transactions, and so forth) you can use a shared property as a counter. Each time you create an instance of the class, you can increment the value of the shared property in its constructor and decrement it in its destructor. (For information about constructors and destructors, see the section "Constructors and Destructors" on page 443.) When you access it, it will give you the current number of instances of the class.

For example, consider the example in the section "Using Classes in Your Projects" on page 450. The local variable "person" stores a reference to the instance of the custom class "Programmer".

```
Dim person as Programmer
person=New Programmer
person.name="Jason"
```

Suppose the Programmer class contains a shared integer property, Total, that gets incremented each time a Programmer instance is created. For example, give the Programmer class a Constructor of:

```
Programmer.Total=Programmer.Total+1
```

The Destructor is:

```
Programmer.Total=Programmer.Total-1
```

Each time a Programmer instance is created or destroyed, the value of the Total shared property is changed to reflect the current count. The value of Total can be accessed from any Programmer instance or from the Programmer class itself.

To create a shared property, do this:

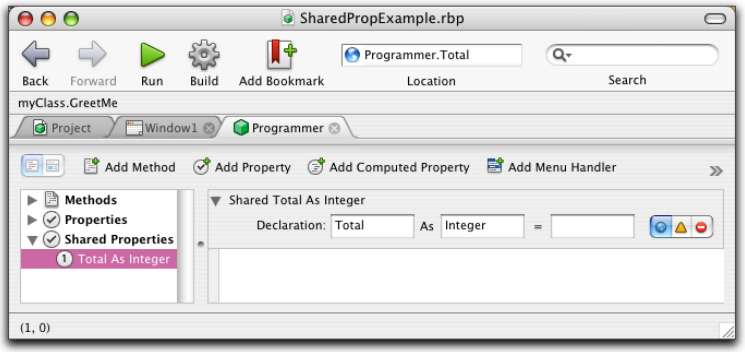
1 **Choose Project ▸ Add ▸ Shared Property or, if it is available, click the Add Shared Property button in the Code Editor toolbar.**

REALbasic displays the declaration area for a shared property. If the Shared Properties item does not already exist, it is added to the Code Editor browser area. (The Add Shared Property button can be added to the Code Editor toolbar by choosing Customize from its contextual menu.)

2 **Declare the property and set its scope the normal way. If desired, set a default value.**

Figure 307 illustrates a shared property in the Code Editor.

Figure 307. A Shared property in the Code Editor.



Adding Constants

You can add constants to classes to store fixed values that the class or the application needs to use. A constant acts like a property but it holds a fixed value for its entire "life." When you create a constant, you give it its value. You can read the constant's value in your code, but you cannot use an assignment statement to change the value of a constant.

Constants that you add to classes have the choices for Scope described in "Scope of a Class's Methods, Properties, and Constants" on page 429.

To add a constant to a class, do this:

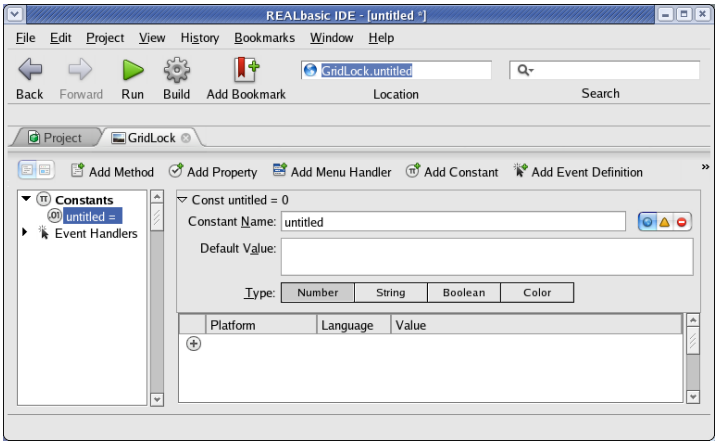
1 **If the Code Editor for the class is not already open, double-click the class's name in the Project pane or click its tab.**

The Code Editor for the class appears.

2 **Click the Add Constant button or choose Project ▶ Add ▶ Constant.**

The Add Constant declaration area appears above the Code Editor area (Figure 308).

Figure 308. The Add Constant declaration area.



3 **Enter the name of the constant and its value.**

4 **Set the data type of the constant by clicking one of the four Type buttons, Numeric, String, Boolean, or Color.**

If you clicked Color, a color patch appears to the right of the color button. Click it to display the Color Picker for your platform and then choose the color from the color picker.

5 **Set the Scope of the constant by clicking one of the three Scope buttons.**

Your choices, from left to right, are Public, Protected, and Private.

Figure 309. The Scope buttons.

For information about Scope, see the section, "Scope of a Class's Methods, Properties, and Constants" on page 429.

6 **(Optional) Use the Localization table at the bottom of the dialog to define different values for the constant for different platforms and language combinations.**

See the section "Using Constants to Localize your Application" on page 307 for details on the Localization table.

When you are finished, the browser area of the Code Editor shows the new constant's name and value, with an icon that indicates its data type.

Adding Methods

You can add methods to classes to provide functionality that the class previously didn't have. For example, in the Gridlock example, there is a method that draws a row by column grid based on the number of rows and columns that is passed to it.

For more information about the options available when you create methods, see the section, "Adding Methods to Windows" on page 264.

To add a method to a class, do this:

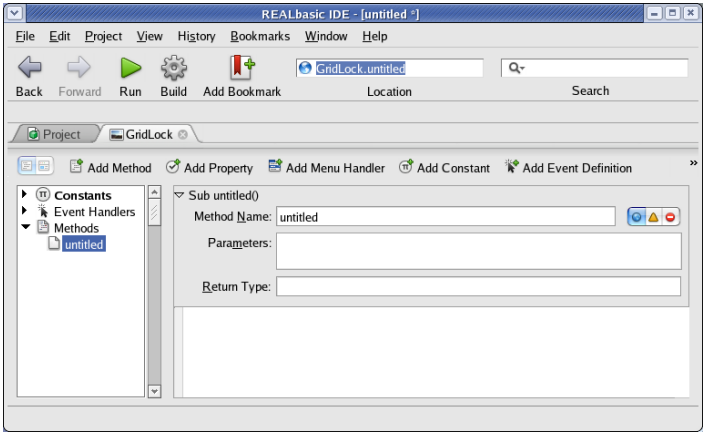
1 **If the class's Code Editor is not already open, double-click on it in the Project Editor to open it or click on its tab.**

The Code Editor for the class appears.

2 **Click the Add Method button or choose Project ▶ Add ▶ Method.**

The Method declaration area appears above the Code Editor area.

Figure 310. The Method declaration area.

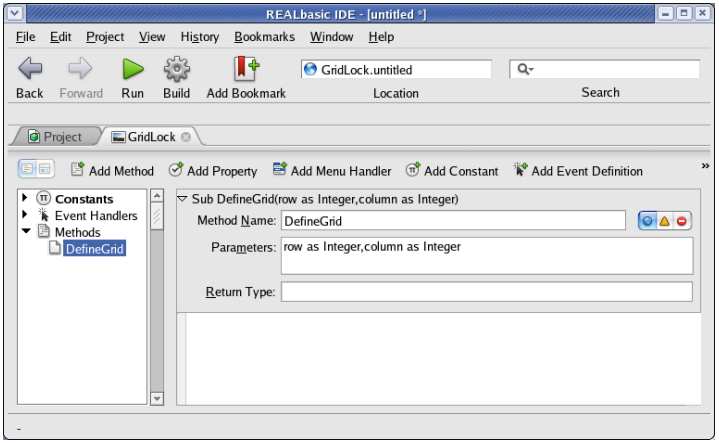


3 **Enter a name for the method.**

4 **Enter the parameters if any, separating multiple parameters with commas, as shown in Figure 311.**

A method does not need to have parameters.

Figure 311. A completed Method Declaration.



There are several advanced features available when you declare the parameters of the method. For more information, see the sections "Passing a Parameter by Value or Reference" on page 270, "Setting Default Values for a Parameter" on page 271, "Setter Methods" on page 273, and "Constructors and Destructors" on page 275.

5 (Optional) If the method will be a function, enter the data type of the value to be returned.

If the value to be returned by the function is an array, place empty parentheses after the data type of the array. For example, to indicate that you will return an array of integers, enter "Integer ()" in the Return Type field.

6 Choose a Scope for the method or function by clicking a Scope icon.

Your choices, from left to right, are Public, Protected, and Private.

Figure 312. The Scope buttons.

For more information on Scope, see the section "Scope of a Class's Methods, Properties, and Constants" on page 429.

When you are finished, the new method is listed in the browser area in the Methods group. If the Scope of the method is Protected or Private a badge appears in the browser and the word "Protected" or "Private" is added to the Sub or Function statement in the declaration area.

Adding Shared Methods

Just as properties can be declared as either instance or shared properties, a method can be declared as either an instance method or a shared method. As is the case for shared properties, a shared method belongs to the class, not an instance of the class. A shared method can be called without instantiating an instance of the class and is also accessible from all instances of the class.

The Self keyword is not available in a shared method or shared computed property and you cannot access instance methods or instance properties inside a shared method unless you are doing so via an instance.

To create a shared method, do this:

1 **Choose Project ▸ Add ▸ Shared Method or, if it is available, click the Add Shared Method button in the class's Code Editor toolbar.**

REALbasic displays the declaration area for the type of item you chose. If the category of item does not already exist, it is added to the Code Editor browser area. (The Add Shared Method button can be added to the Code Editor toolbar by choosing Customize from its contextual menu.)

2 **Declare the shared method and set its scope the normal way.**

3 **Write the shared method in the Code Editor.**

Here is an example that illustrates the difference between instance and shared methods. Create a class, myClass, in the Project Window and create the following simple method of myClass:

```
Sub WelcomeMe(a As String)
  MsgBox a
```

If you create WelcomeMe as an instance method, you can only call it from an instance of myClass, e.g.

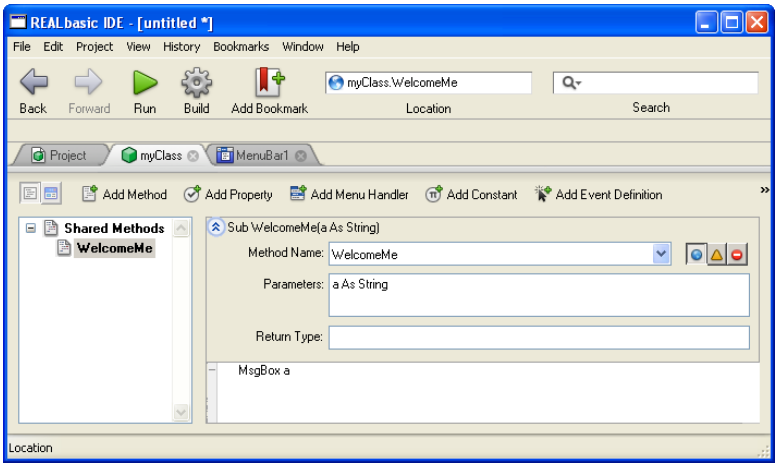
```
Dim greetMe as myClass
greetMe = New myClass
greetMe.WelcomeMe "Hello World"
```

If you create WelcomeMe as a shared method, you can call WelcomeMe with the line:

```
myClass.WelcomeMe "Hello World"
```

Figure 307 illustrates a this shared method in the Code Editor.

Figure 313. A Shared method in the Code Editor.



Adding Event Definitions

Any events in a class you have added code to will not, by default, be available to any instance of the class. Consider this example. You create a class based on the ListBox class and you put some code in its Open event handler. Any instances of that class that appear in a window will not have their Open event handler. The assumption is that since the event handler of the class has code for the Open event, it is handling that event.

There may be times, however, when you want the class to have code in an event handler but you also want to be able to put code in that event handler for an instance of the class. An example of this is when you set up default values. In the Open event handler, you might set the default values of the class. For example, in a class that displays the names of the months in a ListBox, you might want to select the current month name by default. However, when you use this class in a window, you might want to be able to override the default action and choose a different month instead. The instance of the month's ListBox won't have an Open event handler because its class is handling the Open event.

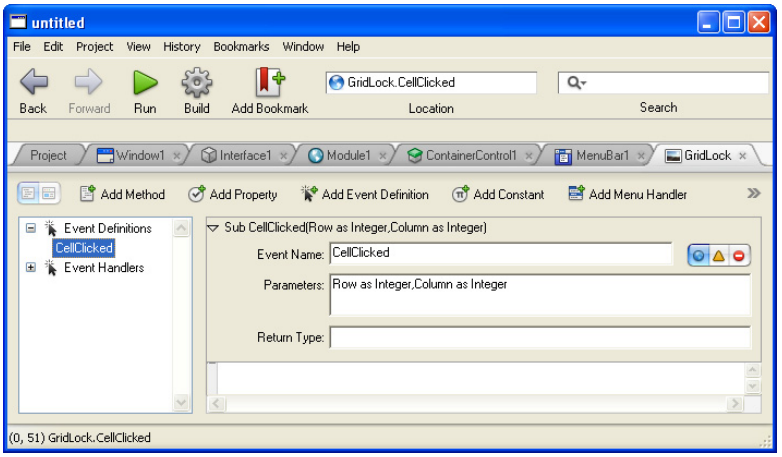
Adding new event definitions solves this problem. You add a new Open event definition to the class and then call it from the class's Open event handler, as if it were a method. New events are available only to the instances of the class. When you add a new Open event, you are adding that event to any instance of the class. When will this new Open event occur? Since you are calling it in the class's Open event handler, it will occur when the window opens — just like a regular Open event handler.

Let's look at another example in which you would want to add new events. Suppose you are creating a custom class that will display a grid. The grid allows the user to click on individual cells to turn them on and off. You might want to add an event that occurs when the user clicks on a cell in the grid. Let's call this event "CellClicked." You also want the event to be passed the row and column numbers

where the click occurred. In any particular instance of the class, you could then use the CellClicked event as a place to take action when the user clicks in a cell.

So how do you go about adding the CellClicked event? First, add a new Event Definition called CellClicked to the class. You want to pass the row and column numbers to this event, so include them as parameters for the event. Figure 314 shows what the New Event declaration area might look like when you are adding the CellClicked event.

Figure 314. The Event Definition declaration area.



The next step is to determine when this event will occur. Since the user clicks the mouse to select a cell, it makes sense that this event is triggered when he clicks the mouse. In each instance of the GridLock class, the CellClicked event is available and is listed in the control's events.

For the Canvas control (the class the grid class would be based on), this means calling this event in the MouseDown and MouseDrag event handlers. To do this, call the CellClicked event as if it were a method. You do the necessary calculations to determine the row and column numbers and pass these to the CellClicked event.

The REALbasic language also has a command that you can use to unambiguously call an event when the event has the same name as a method. Use the keyword "RaiseEvent", followed by the name of the event and any parameters it requires.

When the user clicks on a cell, the MouseDown event handler of the class is executed. Since CellClicked would be called in this event, this causes the CellClicked event to be called and passed the row and column numbers. This causes the CellClicked event to occur for the instance of the class the user clicked on in the window. The class is basically calling a subroutine of the instance of the class. And, because the CellClicked event could be designed to return a value, the instance of the class can return data back to the super class. This could be beneficial in this particular example if you wanted to filter the click. You could code the class to only continue with handling the click should the CellClicked event return False (use False since this is the default value returned by a function). This would allow any

instance of the class to determine which cells are valid for clicking and which cells are not.

See the "Gridlock" project on the REALbasic CD or the REAL Software web site for an example of this kind of event definition.

Extending Classes

You can also add methods to existing REALbasic classes. You do this by adding a *class extension method* to a module. Once added, the new method can be called as if it were built into REALbasic. Class extension methods are added only to modules, but they are called from classes.

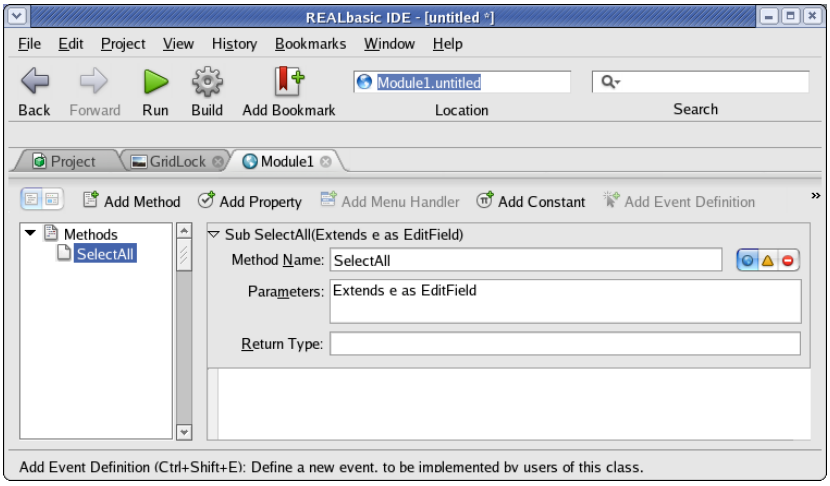
To define a method as a class extension method, use the "Extends" keyword prior to the first parameter in the parameter declaration. The data type of the first parameter is the object type from which the method will be called. In other words, the use of the "Extends" keyword indicates that the parameter is to be used on the left side of the dot (".") operator in a calling statement. When you use the Extends keyword, you do not have to pass an object of that type to the method. You can add "normal" parameters to the parameter declaration that follow the Extends parameter.

Writing a Class Extension Method

For example, you can add a method that can be called from an EditField object. Suppose you want a method that selects all the text in an EditField. You name it "SelectAll". Create a module in the project if it doesn't already have one and add the SelectAll method to the module.

Since the SelectAll method doesn't take any "normal" parameters, the parameter declaration uses only the parameter that takes the Extends keyword. This is shown in Figure 315.

Figure 315. Declaring a class extension method of the EditField class.



The Parameters area shows the declaration:

```
Extends e As EditField
```

The Extends keyword indicates that the method can be called from any EditField object that is in the project. Note also that the Scope for the class extension method is Global.

The code for the SelectAll method is simple:

```
e.SelStart=0
e.SelLength=Len(e.Text)
```

SelStart places the insertion point just before the first character in the EditField and SelLength selects as many characters as are in the EditField.

Calling a Class Extension Method

You can call this method from anywhere in the project. You can simply use the line:

```
EditField1.SelectAll
```

This assumes that there's an EditField in the window named EditField1.

To use the class extension method in another project, all you need to do is copy the module that contains the definition of the class extension method to the other project.

Constructors and Destructors

When you create a new object, you will sometimes want to perform some sort of initialization on the object. The *constructor* is a mechanism for doing this. An object's constructor is the method that will be executed automatically when an instance of the class is created.

Constructors

You add a constructor to a class by adding a method to the class called "Constructor". It will be called automatically when an instance is created via the New operator. If your constructor accepts parameters, you must pass the required number of parameters and they must be of the correct data type.

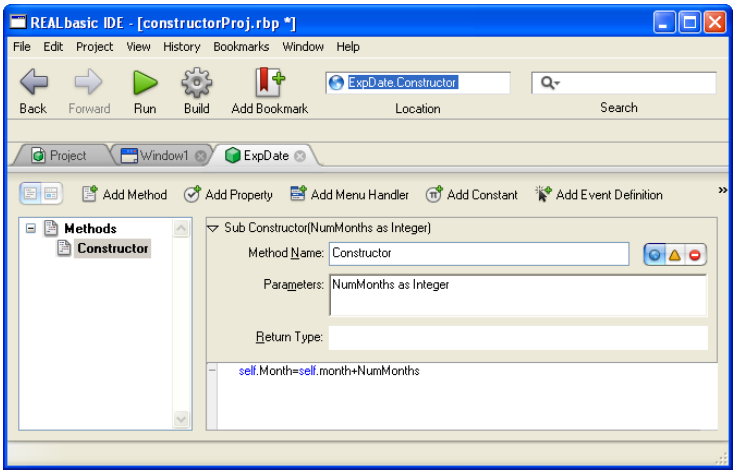
Here is a simple example. Suppose you are managing a service that sells monthly subscriptions. You want a custom version of the Date class the automatically takes the value of the expiration date when an instance is instantiated.

First, add a new class to the project and set its Super class to date and rename it "ExpDate". Double-click it to open its Code Editor and create a new method called Constructor. The constructor takes one integer parameter, NumMonths, the

number of months the customer has signed up for. The constructor has only one line of code:

```
self.Month=self.month+NumMonths
```

Figure 316. The constructor for the expiration date class.



When an instance of a 'regular' Date object is created, it is initialized to the current date, so this line increments the current month number by the number of months that is passed in as a parameter.

When you need to get the expiration date for a customer who signs up for 6 months, you can get it like this:

```
Dim ex as New ExpDate(6)
EditField1.text=ex.ShortDate
```

The number months is passed in as a parameter and the new instance holds the expiration date instead of the current date.

Initializing an instance of a control class

If you need to initialize an instance of a control class, don't use a constructor. Instead, put the code that does the initialization in the Open event of the control. The section "Examples of Subclasses" on page 421 has an example in which a custom ListBox is initialized to display the months of the year and select the current month. This is the functionality of a constructor, but the code instead goes in the control's Open event.

Destructors

You can also create a destructor. The destructor is called automatically when an instance of the parent class is deleted or goes out of scope — for example, when the user closes the window. Name the new method "Destructor". Destructors take no parameters and do not return a value.

Destructors are called when the last reference to an object is removed, even if execution is in a destructor for another object. Note that this means you can cause a stack overflow if your destructor triggers other destructors in a deep recursion. However, such overflow will not happen as long as properties of the object are being cleaned up automatically. So, it is generally preferable to *not* set properties to Nil in your destructor, but instead let REALbasic clean them up for you.

Overloading

REALbasic also supports what is known as *overloading* of methods. A language that supports overloading allows you to have two or more methods with the same name but have a different number of parameters or parameters with different data types. When that method name is called, REALbasic will figure out which method you 'mean' to call from its parameters.

A good example of a built-in overloaded function is the '+' operator. If its arguments are numbers, it computes the sum; if the arguments are strings, it concatenates the strings.

Overloading Custom Classes

Using the language, you can overload these operators so that you can perform arithmetic and comparison operations on your custom classes. For example, if you want to add two objects that store lists, you need to write a function that defines the "+" operator for that custom class. REALbasic has a set of built in keywords that you use for this purpose.

The Operator_*keyword* represents a set of reserved words that enable you to implement the standard arithmetic and comparison operators in your custom classes. The supported operators and keywords are shown below:

Operator	Keyword
+	Operator_Add Operator_AddRight
-	Operator_Subtract Operator_SubtractRight
*	Operator_Multiply Operator_MultiplyRight
/	Operator_Divide Operator_DivideRight
\	Operator_IntegerDivide Operator_IntegerDivideRight
Mod	Operator_Modulo Operator_ModuloRight
And	Operator_And Operator_AndRight
Not	Operator_Not

Operator	Keyword
Or	Operator_Or Operator_OrRight
=, <, >, <=, >=	Operator_Compare
(lookup)	Operator_Lookup
(negation)	Operator_Negate
(convert)	Operator_Convert

For example, to add the ability to add two instances of a custom class, you define a function called "Operator_Add" as a method in the custom class. This function defines the addition process for that class. After it is defined, you can simply use the + operator to add two instances of that class. The + operator will automatically call your custom Operator_Add method

For detailed information about operator overloading for custom classes, see the section on each Operator_ keyword in the *Language Reference*.

Assigning a Value to a Method

When you call one of your methods, you can optionally assign a value to it using the syntax that you normally use to assign a value to a property. That is, you can write:

```
objectname.methodname = value
```

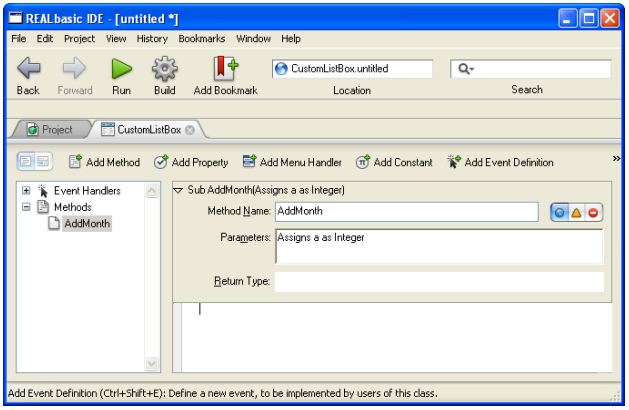
When you do so, the value becomes the value of the last parameter that the method expects. It must be of the same data type as that parameter. If the method expects more than one parameter, you must pass values of all but the last parameter in the normal way. When you want to use this syntax, you must use the "Assigns" keyword when you declare the method. Here is an example.

Suppose you create a custom class based on ListBox called CustomListBox. You can add a method to this class that adds a row to the ListBox based on the number passed to it. The variable, a, is declared as an Integer parameter, as shown in Figure 317.

```
Select case a
  case 1
    addrow "January"
  case 2
    addrow "February"
  case 3
    addrow "March"
  case 4
    addrow "April"
  case 5
    addrow "May"
  case 6
    addrow "June"
  case 7
    addrow "July"
  case 8
    addrow "August"
  case 9
    addrow "September"
  case 10
    addrow "October"
  case 11
    addrow "November"
  case 12
    addrow "December"
End Select
```

When you declare the method, use the "Assigns" keyword in the following way.

Figure 317. Using the "Assigns" keyword in a method declaration.



When you use the Assigns keyword, you pass the integer value to the AddMonth method using the Assignment operator:

```
CustomListBox.AddMonth = 2
```

If the method takes more than one parameter, the value you assign is used as the value of the last parameter. Include all the parameters in the call except the last one.

For example, suppose you write another method of the CustomListBox class called "ChangeCell" that takes *row* and *value* as parameters. *Row* is the cell in the ListBox you want to change and *value* is the new value.

```
Sub ChangeCell(row as Integer, Assigns value as String)
  Cell(row,0)=value
```

If you want to change the value of the third element in the ListBox to "New York", the "normal" way to call ChangeCell is:

```
CustomListBox.ChangeCell(2,"New York")
```

If you use the "Assigns" keyword, you can also call it using *row* as the parameter and assign the new value using an equals sign:

```
CustomListBox.ChangeCell(2) = "New York"
```

Using Arrays of Classes

As is the case with any object, you can create an array of instances of a class. For example, you can create an array of controls as part of your interface to make it easy to manage the controls. You simply give the controls the same name and distinguish among them using its Index property. Control arrays are described in the section "Sharing Code Among An Array of Controls" on page 281.

You can also create an array of instances of non-control classes. When you do so, you can take advantage of the object hierarchy or class interfaces. If myClass1 is subclassed from mySuperClass1 or a class interface implemented by myClass1, then an array of myClass1 can be used whenever an array of mySuperClass1 is expected. The array maintains its original element type and each insert, append, and assignment statement to an array element is checked to make sure that the new value matches the element type. If the type does not match, a TypeMismatchException occurs. You can manage it via a Catch statement in a Try block or in an Exception block.

Casting

An extremely powerful way of creating generic, reusable code is to take advantage of the object hierarchy using *casting*. The idea is best illustrated by an example.

Since the objects you create are subclasses of base classes in the REALbasic language, you can always test to see whether an object is a member of a specific subclass. The IsA operator in the language does this.

With the IsA operator, you test whether an object is of a specific subclass and, if it is, cast it as that type to do something specific with it. You cast an object by using the classname as a function that operates on the instance.

When you cast an instance, REALbasic does not do error-checking for you to guarantee that you are casting to a legal object type. The instance that you are casting has to be of the type that you specify. Casting just tells REALbasic to treat the object as a instance of the class to which it is cast. It doesn't convert it from one class to another.

Here is an example that uses a For loop to cycle through all the controls in a window to test whether each control is an EditField. If it is, it casts the control, gets its name and the values of its Text and DataField properties, and assigns the contents of the EditField to the field in a database table named *fieldname*.

```
Dim r as DatabaseRecord
Dim fieldname, fieldContents as String
.
.
For i = 1 to Self.ControlCount  //number of controls in window
  If Self.control(i) IsA EditField then
    fieldname = EditField(control(i)).DataField //cast it
    //the text property assigned to the contents of that field
    fieldContents = EditField(control(i)).text //cast the control as an EditField
  r.column(fieldname)= fieldContents
  end if
next
```

As you can see, the code does not refer to any specific windows, EditFields, or even databases. Therefore, you can write this routine once and use it in any window in any project that includes a database.

Managing Menus within Classes

Classes that can receive the focus can control the menus when they have the focus. This makes it even easier to encapsulate code within a control. Classes that you create based on classes that can have the focus will have an EnableMenuItems event handler and can have menu handlers for any of the menu items in your project.

When an instance of a class has the focus and the user clicks in the menu bar (or presses a keyboard shortcut for a menu item), the class's EnableMenuItems event handler is executed. This gives the class the opportunity to enable or disable any menu items. The window's EnableMenuItems event handler will be executed next, followed by the application class's EnableMenuItems event handler (assuming you have created a class with Application as its super class). If a menu item is then selected, REALbasic

first checks the class to see if it has a menu handler for the selected menu item. If the menu handler exists, it is executed, followed by the window's menu handler (if it has one for the selected menu item), followed by the application's menu handler (if it has one for the selected menu item).

The SecureEditField class mentioned in the section "Examples of Subclasses" on page 421 is an example of a class controlling menu items. When the SecureEditField has the focus and the user clicks in the menu bar, the SecureEditField's EnableMenuItems event handler sets the Enabled property of the Cut and Copy menu items to False, disabling them. These menu items would normally be enabled automatically by REALbasic.

Another example of a class that manages menus is a class based on the ListBox that allows the user to use the Cut and Copy menu items to move menus between ListBoxes. See the ClipListBox project of the REALbasic CD for an example.

Using Classes in Your Projects

Before you can use a class in your project, you must first understand a few concepts and terms. The use of a class in a project involves three items: the class, the instance, and the reference.

The Class

The class is a template set of events, methods, and properties from which you create subclasses and instances.

The Instance

An instance is a place in memory that stores a copy of the properties of the class. Methods are not stored in memory with each instance. Instead, they are loaded from the class into memory when they are called.

The Reference

The reference is a value stored in a property or local variable that keeps track of where the instance is in memory. You use the property or local variable holding the reference to access the instance of the class. In the following example, "person" is a local variable storing a reference to the instance of the custom class "Programmer." The reference is then used to access the value in the name property of the instance created using the New operator.

```
Dim person as Programmer
person=New Programmer
person.name="Jason"
```

You will learn more about using the New operator later in this chapter.

How you use a class in your project depends on whether the class is based on a control.

Subclasses Based on Controls

The easiest way to add a class based on a control to a project is to drag the control from the Controls list to a Window Editor.

A new class, called *ControlName*X, where X is a sequential number, will be added to the window and the Properties pane will show that it is subclassed from the control's class. Use the Properties pane to rename the new class.

You can also create a subclass of a control class by clicking the Add Class button from the Project Editor or by choosing Project ▶ Add ▶ Class to add a new class to the project and then use the Properties pane to set the Super Class of the new class to a control class.

Since you've subclassed the class from a control, you can then add new properties and methods to the class — to create your own custom control — and then add an instance of the custom control to a window. See the section "Creating Custom Interface Controls with Classes" on page 455 for an example.

To add an instance of a class based on a control to a window, use the Controls drop-down list in the Window Editor to choose Project Controls and then drag the class from the Controls list to the window in which you want the new instance—just as if you were to create an instance of a built-in control by dragging its icon from the Controls list.

Classes Based on Classes Other Than Controls

Classes don't have to be based on controls. You can also create classes based on classes that are not based on the Control class. For example, the Thread class is not based on the Control class. You might need to create a subclass of the Thread class and add properties to it to store information used or created by the thread. You might even need to create classes that have no super class. For example, you could create a class called "People" that had properties like Name, Age, and Height to store information about people. You could then create a subclass of people called "ComputerUsers" which would add additional properties that define a computer user.

To create an instance of a class based on a class other than one of the control classes, you must first have a place to store the instance. You can store the instance in a property or a local variable. The property or local variable must be of the same type as the class or one of the class's super classes. For example, if you need an instance of the Date class to store an individual's birth date, you create a variable of type Date to store a reference to the instance.

You use the Dim statement to create the variable—just as you would create a variable that stores a REALbasic data type. Use the New operator to create a new instance of the class in memory and then assign a reference to the new instance to the property or local variable you have typed.

```
Dim birthDate as Date
birthDate=New Date
```

In this example, the local variable "birthDate" is typed as class Date. The New operator is then used to create a new instance of Date and assign a reference to this variable.

You can use a simpler syntax. You can place the New operator in the Dim statement as a modifier. When you do so, the statement creates the local variable and the reference in one step:

```
Dim birthDate As New Date
```

Accessing the Properties and Methods of a Class

Once you have created an instance of a class and stored a reference to it in a local variable or property, you can access its properties and methods the same way you access any object's properties and methods. For example, in the code below, the local variable birthDate now has access to the properties of the Date class. You can use them to assign values to properties.

```
Dim birthDate as New Date
birthDate.Year=1967
birthDate.Month=3
birthDate.Day=21
```

In this simple example, the instance is based on one of REALbasic's built-in classes. If it was based on one of your own custom classes, which in turn was based on a built-in class, the instance would have access to the properties and methods of both its own super class and the built-in class that its super class was based on.

When are Instances of Classes Removed From Memory?

REALbasic manages memory for you automatically using something called *reference counting*. REALbasic maintains a counter for each instance that you create and records references to the object. Each reference increments the counter and removing a reference to the object decrements the counter. You remove a reference, for example, by setting it to Nil or by closing the window in which the instance is located. When the reference counter reaches zero, the object is removed from memory immediately.

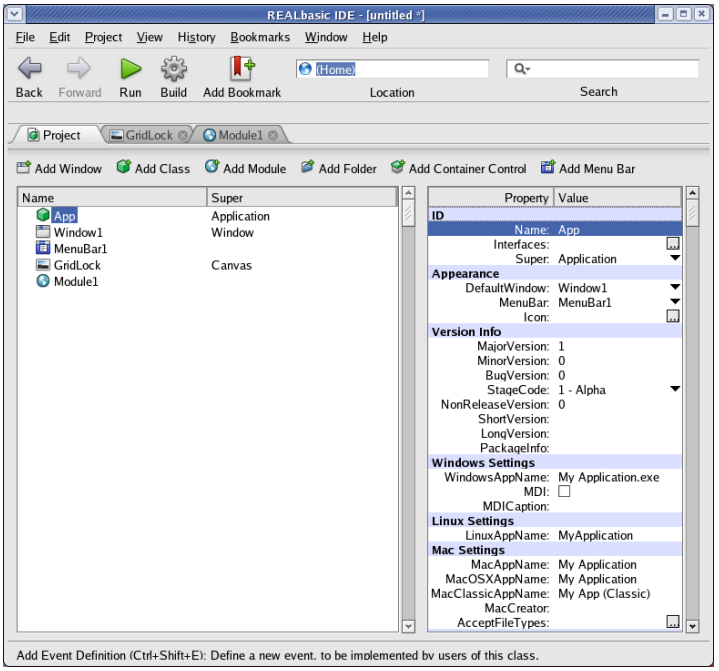
This means that instances of classes are removed from memory automatically when they are no longer used. Suppose you create a class based on a ListBox. You then create an instance of that class in a window. When the window is opened, the instance of the class is created in memory automatically. When the window is closed, the instance of the class is automatically removed from memory. If you store the reference to a class in a local variable, when the method or event handler is finished executing, the instance of the class is removed from memory. If you store a reference to an instance of a class in a property, the instance will be removed from memory when the object owning the property is removed from memory.

The Application Class

The Application class is used to create a subclass that represents your application rather than a window or a control. When you create a new project, a subclass based on the Application class is added to your project automatically. This is the "App" class that is listed below the default menubar in the Project Editor. Its Properties pane shows that it is derived from the Application class and the default menubar is assigned to the application as a whole.

If you wish, you can create more than one subclass based on the Application class. However, there is only one such subclass that plays the role of the "blessed" App class that has the built application's properties in the Properties pane of the Project editor.

Figure 318. The default App class.



Special Event Handlers

The Application class has special event handlers. They are:

- **Open**: Executes when the you run the application by clicking the Run button in the Toolbar or by choosing Project ▶ Run (Ctrl+R or ⌘-R) or when launching a standalone version of your application.
- **Close**: Executes when you quit your application either from the Debugging environment or in a stand-alone application.

- **NewDocument**: Executes when the stand-alone version of the application is launched without double-clicking one of the application's documents.
- **OpenDocument:** Executes when one of the application's documents is double-clicked from the desktop.
- **EnableMenuItems:** Executes when the user clicks in the menu bar but before any menu items are displayed. The EnableMenuItems event handler executes after the EnableMenuItems event handler of any classes with instances in the frontmost window and after the window's EnableMenuItems event handler. This is the event handler that should be used to enable menu items that should be enabled regardless of whether there is a window open or not (This possibility exists on Macintosh applications). Note that if a menu item should always be enabled, you should use its AutoEnable property instead of an EnableMenuItems event handler.
- **HandleAppleEvent:** Executes when an AppleEvent is received by the application.
- **Activate:** The application is being activated. This occurs when the application is opening and when it is being brought to the front.
- **Deactivate:** The application is being deactivated. This occurs when another application or a desktop window is being brought to the front or when the application quits.
- **Unhandled Exception:** Executes when a runtime error occurs that is not handled by an Exception Block. This event gives you a "last chance" to catch runtime errors before they cause your application to quit "unexpectedly." For more information on runtime errors, see the section "Runtime Exception Errors" on page 510.

Scope of the App Class's Properties

When you add properties, methods, and constants to the App class, you declare the Scope of the new item. Your choices are Public, Protected, or Private. For descriptions of these choices, see the section "Scope of a Class's Methods, Properties, and Constants" on page 429.

Public properties of the Application class are accessible to all code in your project. That is, if you want to access the property anywhere in your application, set its Scope to Public when you declare the property.

Use the App function to access the App class's Public properties outside the App class. For example, to access the a property called "Separator" outside the App class, use the syntax:

```
App.Separator
```

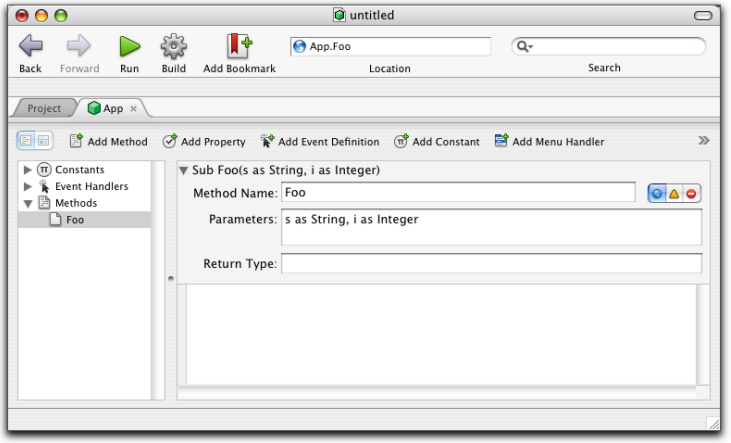
To access an App class property in the Properties pane of a window or class outside the App class, precede the reference by the number sign, "#". In this example, you would enter "#App.Separator" to access this property in the Properties pane.

If you set a property's Scope to Protected or Private, it can be used only within the App class.

Scope of the App Class's Methods

Public methods of the Application class are accessible to all code in your project. If you want to access a method outside of the App class's own methods and event handlers, be sure to set its Scope to Public. For example, if you add a method to the App class called "Foo", set its Scope as shown in Figure 319.

Figure 319. Creating a Public method of the App class.



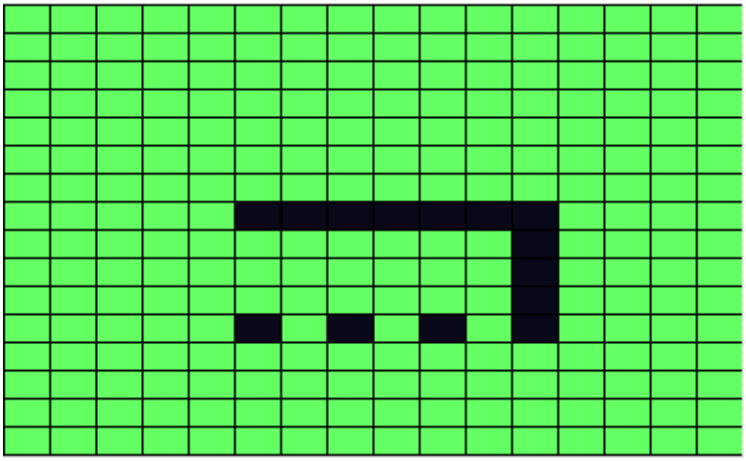
If you set a method's Scope to Protected, then you can call it only within the App class's own methods and event handlers. Since this method is public, you can call it the App class with the line:

```
App.Foo
```

Creating Custom Interface Controls with Classes

One of the most important uses of classes is for creating custom interface controls. While REALbasic provides most of the interface controls you will need in your project, you may find you need to create interface controls that are not built-in to REALbasic. Suppose you need to create a control that displays a grid of cells. You want the user to be able to click on the cells in the grid to select them. Figure 320 shows an example of what such a grid control might look like.

Figure 320. The custom grid control.



Custom interface controls are created by building a subclass based on the Canvas control. The Canvas control gives you an area in which you can draw your control and it receives events allowing you to interact with the user. For example, in the grid control example in Figure 320, the object is an instance of the Gridlock class, which was derived from the Canvas class (See Figure 300 on page 427). This is the Gridlock class example that you can find on the REALbasic web site. It inherits all the properties and events of the Canvas control and has special methods, properties, and events that enable it to present and operate this interface.

The Paint event handler of the Gridlock control is used to draw the grid. The Gridlock class has properties that store the number of rows and columns the programmer wants for a particular instance of the Gridlock. There are also properties that store the selected cell color and the unselected cell color. When the user clicks in the grid area, the MouseDown event handler for the Gridlock class executes. The code for this event handler determines which cell was clicked and then determines if the cell should now be selected or unselected. A new event called CellClicked has been added to the Gridlock class that is executed when the user clicks on a cell. The purpose of this event is to allow an instance of a Gridlock class to react to a cell click. The CellClicked event handler is passed the row and column numbers of the cell that was clicked. The CellClicked event handler also acts as a function. If an instance of the Gridlock class returns True in the CellClicked event handler, the Gridlock class assumes the programmer wants to filter the click, so it acts as if the user didn't click in the cell.

Drawing Your Custom Control

The Paint event handler of a Canvas control (or a Canvas control-based subclass) is executed any time the control needs to be redrawn. For example, if a window is covering part of the control and it is then moved to uncover more of the control, the Paint event handler executes to redraw the control. If the look of the control doesn't change at all when it's used, you can do all of the drawing of your control in the

Paint event handler. However, if your control changes, you will need to take a different approach. For example, the Gridlock control changes when the user clicks on a cell. The Gridlock control also has a method that allows the number or rows and columns in the grid to be changed on the fly. This requires the grid to be redrawn.

In the Gridlock example, the grid needs to be redrawn at two different times. It needs to be redrawn in the Paint event handler when something like a window positioned over the control has uncovered a portion of the control, and when the grid is redefined to have a different number of rows and columns. Because of this, the code to do the actual drawing is placed in its own method. The method is called DrawGrid and it is passed the Graphics property of the Canvas control that the Gridlock class is based on. The DrawGrid method can then use this property to redraw the grid. By placing this code in a separate method, the same code can be used by the Paint event handler and by the DefineGrid method. The Paint event handler is passed a reference to the Graphics property of the Canvas so this reference can be passed on to the DrawGrid method when calling it from the Paint event handler. The DefineGrid method calls the DrawGrid method as well since the grid is being resized and needs to be redrawn. The DrawGrid method can be passed the graphics property in this case by using the syntax:

```
DrawGrid Me.Graphics
```

Me is a reference to the instance of the class in the window. So although this code is being called from inside the Gridlock class, the use of Me allows it access to properties of the instance in use.

Virtual Methods

Virtual methods provide a way for a subclass to have its own version of a method that its super class has. Ordinarily, a subclass inherits the methods belonging to its parent.

Other object-oriented languages sometimes refer to the concepts behind virtual methods and the following topic, as *polymorphism*.

When a subclass has a method that has the same name as its parent, the subclass's version is called unless you use the syntax:

```
parentclassname.methodname
```

To create a 'virtual' method, do this:

1. **Create a class.**
2. **Add a method to the class.**
3. **Create a subclass of the first class.**

4 Add a method to the subclass with the same name as the method you added in step 2.

When the subclass calls the method, it will call its own version and not its parent class's version.

Class Interfaces

A class interface is a construct that you can use to tie together classes that do not share a super class but have something in common in your application. A class interface operates as a "spec" that contains a list of methods that custom classes in your project use. It does not actually contain any code for the methods themselves.

The methods in the class interface are actually placeholders for methods that are actually contained in each custom class that 'implements' the class interface. Also, a custom class can implement more than one class interface.

The term 'implement' simply means that the class has methods of the same names and declarations that are found in the class interface. The class interface specifies the methods and their declarations but not the code.

When you specify that a class implements a class interface, the class must implement all the methods in the class interface and the method declarations must match. However, the classes are free to implement the methods in different ways. For example, a method that changes the font in a StaticText would be implemented in a different way than a method that changes the font in a Canvas control that displays text via calls to the Graphics class.

The process involves three basic phases:

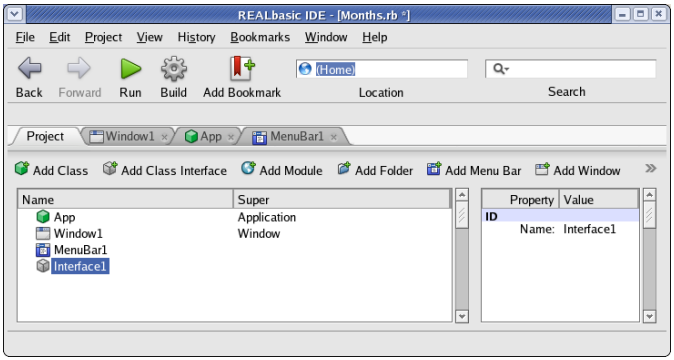
- Creating the class interface,
- Creating the classes that implement the class interface,
- Adding the classes to your project and calling the class interface methods in your program. Typically, that means writing generic code that tests whether a class implements a class interface and executing class interface methods where appropriate.

To create a class interface, do this:

1 If the Project Editor is not displayed, click on its tab and then click the Add Class Interface button or choose Project ▸ Add ▸ Class Interface.

A new class interface is added to the Project Editor. Its icon is hollow, indicating that it doesn't actually hold code. The Properties pane shows that the only property that you can modify is its name. It has no Super Class, as it is not part of the object hierarchy.

Figure 321. The Project Editor with a new class interface.



2 **If desired, use the Properties pane to change the name of the class interface.**

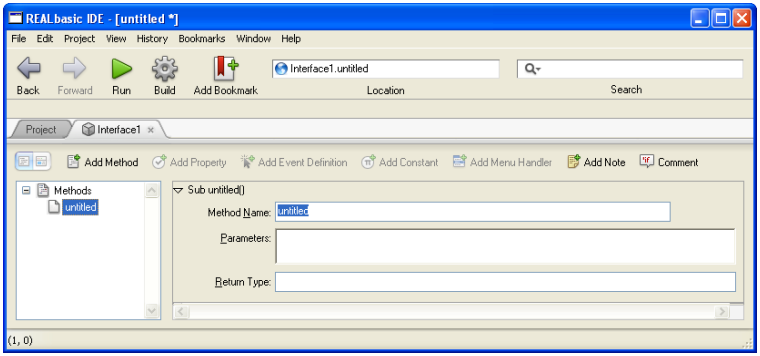
3 **Double-click the Class Interface item in the Project Editor to display the Code Editor for the class interface.**

The Code Editor for a class interface has items only for methods and notes. You cannot create properties or constants for a class interface.

4 **Click the Add Method button or choose Project ▸ Add ▸ Method to add a method declaration to the class interface.**

The Method declaration area appears above the Code Editor area.

Figure 322. The Method declaration area for a Class Interface.



5 **Enter the name of the method, its parameters, and, if it will return a value, the data type of the value being returned.**

In other words, declare the method in the normal manner. The only difference is that there is no way to specify the Scope of a method in a class interface. This is because the code for the method does not live in the class interface at all. You use the Method declaration only to provide the 'spec' for the methods that will be written (a.k.a., "implemented") elsewhere. The method will have several implementations, one in each class that implements the class interface.

For more information on declaring a method, see the section "Adding Methods to Windows" on page 264.

6 Repeat steps 4 and 5 for each method or function declaration of the Class Interface.

After you have created your class interface, you must 'hook it up' to one or more custom classes. To be non-trivial, we assume it will be two or more custom classes. You add the code for the methods declared in the class interface in *each* class that implements the class interface.

To implement a class interface, do this:

1 In the Project Editor, select the class to which you want to add the class interface or, if it does not yet exist, click the Add Class button in the Project Editor or choose Project ▸ Add ▸ Class.

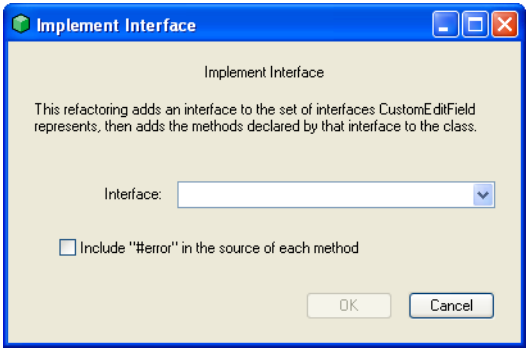
Notice that the Properties pane for the class contains a field for specifying the class interface (or interfaces) for the custom class.

You can specify the class interface or interfaces that the class implements by entering their names in the Interfaces field in the Properties pane or via the Project pane's contextual menu. If desired, you can specify more than one class interface for the class.

2 In the Properties pane's Interfaces field, click on the ellipsis (the box on the right with three dots in it) or right+click (Control-click on Macintosh) on the name of the class in the Project Editor and choose Implement Interface.

The Implement Interface dialog box appears.

Figure 323. The Implement Interfaces dialog box.



3 Choose the class interface you wish to add from the Interfaces drop-down list. If desired, choose the "Include #error" option.

When you do so, the name of the class interface is added to the Interfaces field in the class's Properties pane. REALbasic also adds an Interfaces column to the Project Editor shows the name of the newly added interface. This is shown in Figure 325 on page 462.

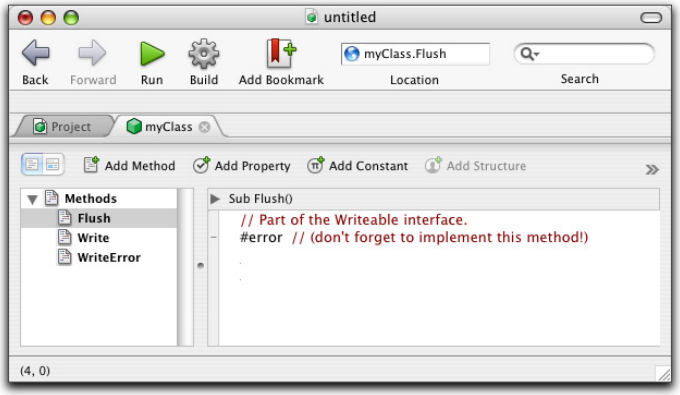
When you choose the class interface, REALbasic adds all the method declarations for the interface to the class's Method Editor. The class's Method Editor then displays the first such method, ready for you to write the method. Each method that is generated by the Implement Interface dialog has a comment line that explains which Class Interface the method belongs to.

If you select the "Include #error in the source of each method" option in the Implement Interfaces dialog, it also includes an uncommented line with the directive "#error". This line causes the compiler to generate a syntax error. The purpose of the line is to remind you to implement the method. If it weren't there and you forget to implement the method, you would satisfy the technical requirement that the method exists, but it would be an empty method.

When you finish implementing the method, you should remove or comment out this line.

Here is an example method that was generated by the Implement Interface dialog.

Figure 324. The Flush method of the Writeable class interface.

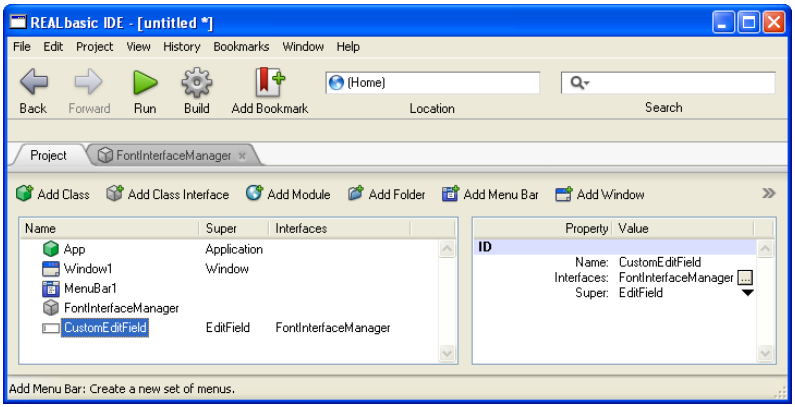


A class can implement more than one class interface. If this is the case, you can choose the Implement Interface contextual menu command again and choose the next interface.

You can also enter the names of class interfaces directly into the Interfaces field in the Properties pane. Click in the text area of the Interfaces field to get an insertion point.

Enter the name of the next class interface into the Interfaces field of the Properties pane, separating the names of the class interfaces by commas. However, this is the non-prefered way to add interfaces because REALbasic does not populate the class's Methods group with the methods specified by the interfaces you select. You must take care to implement all the methods yourself.

Figure 325. A new class that implements a class interface and its Properties pane.



Notice that the Project Editor now has a third column and it lists the class interfaces for the projects' class interfaces.

Implementing Methods

The next task is to implement the methods of the class interfaces that you added to the class. If you used the Implement Interfaces dialog box, you have a head start on this task because it adds the method names and declarations of each class interface to the class automatically. If you entered the names of the class interfaces manually, you now need to add the required methods to the class. Follow the following steps.

4 **If the methods are not already added, click the Add Method button or choose Project ▶ Add ▶ Method.**

The the Add Method pane appears.

5 **Enter a method name and the declaration that was defined in the class interface and click OK.**

This time, the Code Editor supports code entry.

6 **With the Code Editor, write the implementation of each class interface method for this class.**

7 **Repeat steps 4 to 6 for every method declaration in each class interface that is implemented by this class.**

8 **Repeat the entire process (steps 1 to 7) for each class that implements the class interface.**

For example, in Figure 327 two classes implement the class interface, FontInterfaceManager, shown in the Project Editor. Different classes can implement the same method in different ways.

Modifying and Deleting Interfaces

If you change your mind and want to delete or replace an interface, you do so via the Interfaces field in the Project Editor.

Click in the Interfaces field to get an insertion point and then select the name of the interface you want to modify or delete. Type to make the desired changes.

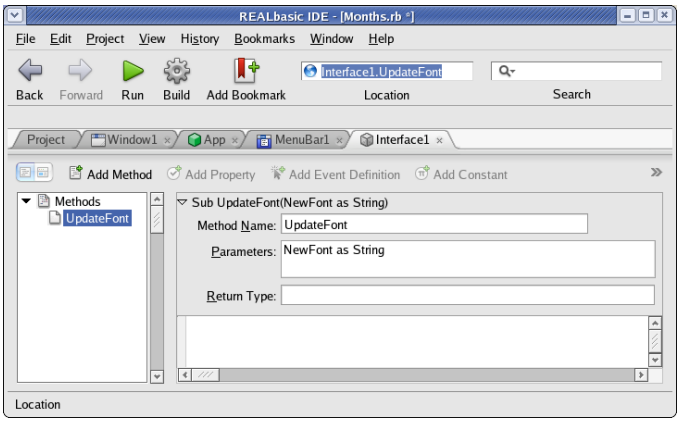
Please note that any methods that were added to the class while the interface was implemented are not modified or deleted when you remove the interface that they belonged to. That is, if you add an interface via the Implement Interfaces dialog, the methods that are specified by that interface remain as part of the class even if you delete the interface itself. You must take care of any "clean up" activities.

A Class Interface Example Project

The following very simple example illustrates the basic concepts. The application has custom classes based on the EditField and StaticText classes that implement a class interface. The class interface has one method specification for updating a font. The user chooses a font from the popup menu and a message is sent to all controls in a window that implement the class interface.

The class interface, FontInterfaceManager, contains one method specification:

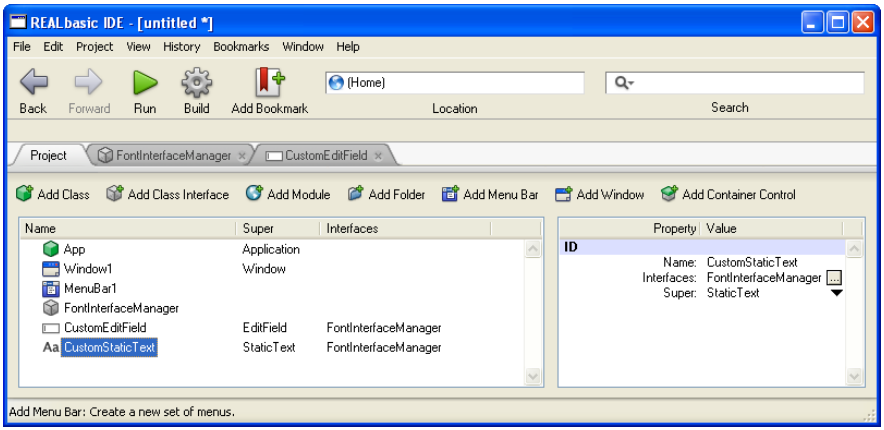
Figure 326. The UpdateFont method specification.



This method declaration area specifies that each class that implements this class interface has to have a method called UpdateFont and it must take one parameter as a string.

Two custom classes based on EditField and StaticText implement this method, as is shown in the Project Editor:

Figure 327. The Project Editor with two classes that implement the class interface.



The implementation of the UpdateFont method happens to be the same in both classes, but this is not a requirement of class interfaces in REALbasic. It is simply:

```
Sub UpdateFont (s As String)
 Self.TextFont=s
```

With these objects in place, generic code can be written that determines whether a control in a window implements the FontInterfaceManager class interface.

Suppose several instances of CustomEditFields and CustomStaticTexts are added to a window. The following code in the Change event of a PopupMenu changes the font displayed by all such controls based on the user's selection. The IsA operator tests whether a control implements the class interface. If it does, the UpdateFont method of the class interface is called, as *implemented by each custom class*. The term Me.Text contains the current selection of the PopupMenu.

```
Dim i as Integer
For i=0 to Self.ControlCount  //number of controls in the window
 If Self.Control(i) IsA FontInterfaceManager then
  FontInterfaceManager(Self.Control(i)).UpdateFont(Me.Text)
 End if
Next
```

As you can see, you can specify that any custom class implements a class interface, regardless of its super class. In this way, class interfaces can group together classes that are not related to one another via the object hierarchy and give them functionality that can be managed in one place in your project.

Creating a new Class Interface from an Existing Class

If an existing class in your project contains methods that you want to reuse as a class interface, you can generate the new class interface from the Project Window.

To use an existing class as the basis of a class interface, do this:

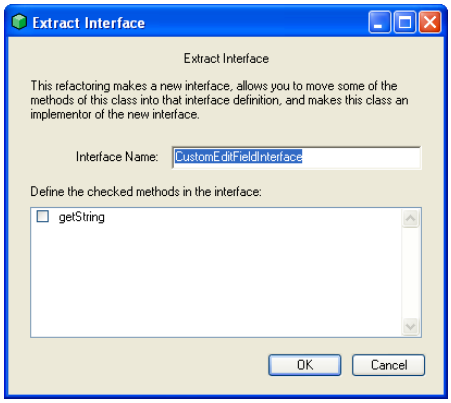
1 **Right+click (Control-click on Macintosh) on the class you want to use as the starting point for the new class interface.**

The contextual menu for the class appears.

2 **Choose Extract Interface from the contextual menu.**

The Extract Interface dialog box appears.

Figure 328. The Extract Interface dialog box.



It lists all the methods in the current class in the ListBox. With this dialog box, you can select some or all of the existing methods of the class by checking the desired methods shown in the ListBox.

3 **Enter the name of the new class interface in the Interface Name field.**

The name of the class you clicked on is entered by default.

4 **Click the checkbox for each method you want to include in the new class interface.**

5 **When you are finished, click OK to save the result.**

REALbasic creates the new class interface, adds it to the project, and makes the current class an implementor of this class interface. That is, the name of the new class interface is now shown in the Interfaces field of the class's Project pane.

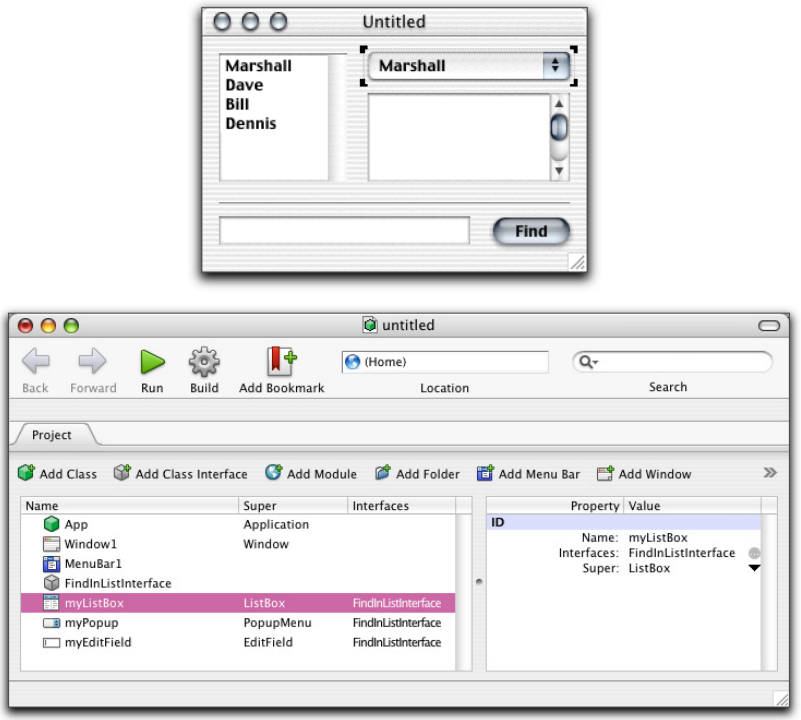
Interface Inheritance

Although Interface Inheritance sounds complicated when described in abstract language, it actually addresses a simple problem. If you have several controls that need to perform the same task but in a different way (depending on the specifics of the types of control) you can write and execute interface-specific code in an elegant way.

Figure 329 on page 466 shows an application that uses interface inheritance. The purpose of this application is to conduct a search for a user-entered string and find the string in the three controls located above the separator: The EditField, ListBox, and PopupMenu are all based on custom classes. Although the task is identical (a find operation), it cannot be done with exactly the same code for all three objects since the three objects store and manipulate data differently. Therefore, each custom class has its own implementations of the methods used to do the search.

The ListBox, EditField, and PopupMenu are all derived from custom classes that use a custom interface, FindInListInterface. They all have a Find function that takes the same parameter, but all implement it differently. The code for the Find button can call all of their Find functions using the same syntax.

Figure 329. An example application that uses interface inheritance.



The user enters a search string in the EditField, FindValue, to the left of the Find button. When he clicks the Find button, the following code is executed:

```
FindIt FindValue.text, listBox1
FindIt FindValue.text, popupMenu1
FindIt FindValue.text, editField1
```

The same method, FindIt, is called for each of the three controls, but each line of code actually executes a different version of FindIt—the one that is appropriate for

that type of control. The second parameter is the name of the control; each control inherits methods from the custom class on which it is based.

The EditField, PopupMenu, and ListBox are all instances of custom classes. The custom classes have two methods, Find and SelectRow, that implement the correct search routines for that object type. This is shown in Table 35.

Table 35: Find functions and SelectRow methods for different controls.

Control	Find Function	SelectRow Method

EditField	Function Find (FindValue as string) as Integer
	dim rows, foundPos, foundCRpos as integer
	rows=-1
	foundPos=instr(text,findValue+chr(13))
	do until foundCRpos>=foundPos
	
	foundCRpos=instr(foundCRpos+1,text,chr(13))
	rows=rows+1
	loop
	return rows
		Sub SelectRow (Row as Integer)
		dim counter, startPos, endPos as integer
		do until counter=row
		startPos=instr(startPos+1, text, chr(13))
		if startPos <> 0 then
		counter=counter+1
		end if
		loop
		endPos=instr(startPos+1,text,chr(13))
		selStart=startPos
		selLength=endPos-startPos

ListBox	Function Find (FindValue as string) as Integer
	dim i as integer
	for i=0 to listcount-1
	if list(i)=findValue then
	return i
	end if
	next
		Sub SelectRow (Row as Integer)
		listindex=row

PopupMenu	Function Find (FindValue as string) as Integer
	dim i as integer
	for i=0 to listcount-1
	if list(i)=findValue then
	return i
	end if
	next
		Sub SelectRow (Row as Integer)
		listindex=row

The FindIt method itself uses the FindInListInterface:

```
Sub FindIt (findValue as String, source as FindInListInterface)
 dim row as integer
 source.selectRow source.find(findValue)
```

The FindInListInterface class simply has two blank methods, Find and SelectRow. It simply defines the methods and their parameters. When the FindIt method runs,

it actually executes the versions of the methods that are appropriate for the control passed as the second parameter to FindIt.

Custom Object Bindings

Object binding allows you to add functionality to your interface without writing any code. There are many binding actions you can choose that are built-in to REALbasic. Built-in binds are listed in the section "Object Binding" on page 141. You can also program your own binds using custom classes. Your custom binds will appear in the Visual Bindings dialog box along with the built-in bindings.

You can drop your custom binding into any project and it will work as if it was built-in to REALbasic. This section shows two examples of custom object binds.

A "Delete All Rows" Bind

This first example creates a user-defined bind between PushButtons and ListBoxes. It allows the user to create a PushButton that deletes all rows from a ListBox. You first create a custom class that implements the binding Interface.

To create the custom bind:

1 **Add a new class to your project.**

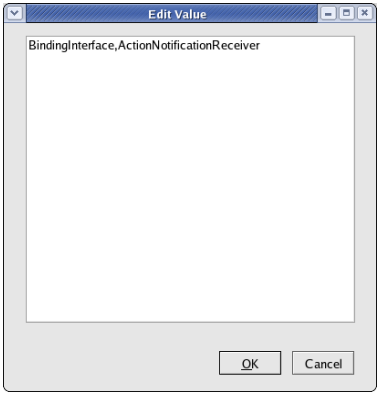
2 **Name this class "DeleteAllRowsBind".**

Since this class is going to be a bind, it needs to support the bindingInterface. Also, you will want to assign an action to the Action event of the PushButton that, when clicked, tells the ListBox to delete all the rows. In order to add code that will execute when the user clicks the PushButton, you will need to support the ActionNotificationReceiver interface.

3 **To support the bindingInterface and ActionNotificationReceiver interface, enter it in the Interfaces property of the class.**

To do so, click on the Text box icon in the Interfaces' Value area to display an alert box with a large text field.

Figure 330. The Interfaces text entry area.



Now that the class supports the bindingInterface, you can add the Bind method.

4 Add a new method to the class and call it "Bind". Include the parameters "Source as Object, Target as Object".

Since the code that executes when the user presses the PushButton will need to know which PushButton and ListBox have been bound, these will need to be stored as properties in the class.

5 Add the following properties to the class: BindSource as PushButton button and BindTarget as ListBox.

6 Add the following code to the Bind method:

```
#pragma bindingSpecification PushButton,ListBox,"Delete all rows in %2
when %1 is pushed"
BindSource=PushButton(source)
bindTarget=ListBox(target)
BindSource.addActionNotificationReceiver self
```

The #pragma statement defines what will appear in the Bind dialog box when the user binds the two objects. It indicates the source class, the target class, and the text that will appear in the Bind window. %1 will be the name of the source control and %2 will be the name of the target control.

When the bind occurs, the controls that are bound will be passed to the Bind method. However, they are passed as generic objects. In order for the code to store them in the BindSource and BindTarget properties you just added to the class, these two object parameters have to be recast as PushButton and ListBox. Lines 2 and 3 in the Bind method do just that.

Finally, the last line connects the bind class to the Action event of the PushButton by calling the addActionNotificationReceiver method of the PushButton class. Self is passed as a parameter to this method and represents the class itself.

Now you need to add the code that will delete the rows when the user clicks the PushButton.

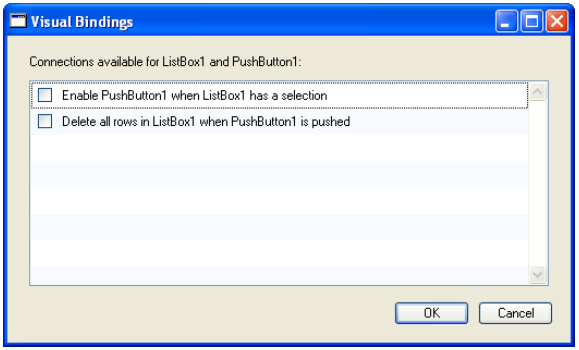
7 To do this, add a method called "PerformAction" to the class.

The code for the PerformAction method is simple:

```
bindTarget.DeleteAllRows
```

To try out this new bind, drag a ListBox and a PushButton onto a window and add some rows of data to the ListBox. Bind the two controls by Command-Shift dragging from the PushButton to the ListBox. When the Bind dialog appears, the custom binding appears in the list, as shown in Figure 331.

Figure 331. The Custom Binding added to the list of binds.



8 Choose the option "Delete all rows from ListBox1 when PushButton1 is pushed" and then test the binding in the Debugging environment.

A "Delete Selected Row" Bind

To create a bind that will delete the selected row from a ListBox when a PushButton is pushed, create another class in your project and duplicate all the steps you did above.

Since this version of the bind should work only when a row is selected, you will need to support a few more interfaces in order to detect when the user selects a row.

1 Add a class called DeleteSelectedRowBind.

2 Add listSelectionNotificationReceiver to the list of interfaces in the Interfaces property of the class.

Add a new method to the class and called it "Bind". Include the parameters "Source as Object, Target as Object".

3 Enter the following code:

```
#pragma bindingSpecification PushButton, ListBox,"Delete the selected
row in %2 when %1 is pushed"
BindSource=PushButton(source)
bindTarget=ListBox(target)


BindTarget.addListSelectionNotificationReceiver self
BindSource.addActionNotificationReceiver self
bindSource.enabled=(bindTarget.listindex >= 0)
```

The first new line links the bind class to any change to the list selection. The last line, sets the source (the PushButton) to enabled if a row of the target (the ListBox) is selected and disables the source if no row is selected when the window opens.

Now you need to add the code that will disable or enable the PushButton once the window is open and the user begins selecting or deselecting rows in the ListBox. In

order to do this, you will need to add the methods that are part of the ListSelectionNotificationReceiver interface to the class.

4 **Add methods "SelectionChanged" and "SelectionChanging" to the class. Neither has any parameters. Add the following code to the SelectionChanged method:**

```
bindSource.enabled=(bindTarget.listindex >= 0)
```

Any time the selection in the ListBox is changed, the bind will be notified and the SelectionChanged method will fire.

The SelectionChanging method has no code.

Lastly, you need to add the code that will delete the selected row when the user clicks the PushButton.

5 **Add the PerformAction method and enter the following line of code:**

```
bindTarget.removeRow bindTarget.listindex
```

Now you can add another button and bind it to the ListBox.

6 **Choose the new bind as the bind action and test it in the Runtime environment.**

Importing Classes From Other Projects

Since classes can be exported, they can also be imported. When a class is exported, it appears on the desktop with a cube icon. Figure 332 shows an example of an exported class. To import a class, just drag the class file into your Project Editor or choose File ▶ Import and use the open-file dialog box to choose the desired class file. This copies the class into the Project. If you wish, you can delete the class file if don't need to use it elsewhere. The Project is not dependent upon it.

Figure 332. An exported class file (Windows).

If a class you are importing is based on another class, that other class must be present in order for the class you are importing to function. If that class is based on one of the built-in classes (like the EditField for example), this isn't an issue. However, if the class is based on a class that isn't built-in, then that other class must be present in your Project Editor.

Exported classes can be encrypted. This means that, while you can import and use the class, you cannot view or edit the source code for the class. If the class is encrypted, a small key badge appears in the class's icon in the Project Editor.

For more information, see the following section "Encrypting Your Source Code" on page 472.

Importing External Project Items

If you need to use the class in more than one project, you can add it to the project as an external project item. An external project item is stored on disk and is referenced by each project that uses it. If a change to the class is made from one project, those changes are made available to all the other projects that reference the external item. Changes to the external project item are saved to disk when you save the project.

To add an external project item to a project, hold down the Shift+Ctrl keys on Windows and Linux or the ⌘ and Option keys on Macintosh while you drag the item from the desktop to the Project Editor. In the Project Editor, the external project item's name will be shown in italics.

For more information, see the section, "External Project Items" on page 428.

Exporting Classes For Use In Other Projects

Classes can be easily exported from your Projects for use in other projects. You can export the class by clicking on the class to select it in the Project Editor and choosing File ▶ Export Class. To export the class as an external project item, follow the steps in the section "External Project Items" on page 428.

Encrypting Your Source Code

You may want to share classes with other REALbasic users. If you wish to share a class with other users but you don't want to share the source code itself, you can encrypt the class before you export it. This creates an exported class that can be imported and used but cannot be edited. The user cannot even view the source code. This is especially important if you plan to create sophisticated classes that you wish to sell as third party add-ons to REALbasic.

Encryption and decryption are not supported in the SE version of REALbasic. Source code protection is supported in both the Standard and Professional versions of REALbasic.

To encrypt a class, do this:

1 **Click on the class to be encrypted in the Project Editor and choose Edit ▶ Encrypt or Right+click on Windows and Linux (Control-click on Macintosh) on the class in the Project Editor and choose Encrypt from the contextual menu.**

You can optionally add an Encrypt button to the Project Editor toolbar. If you have done so, you can encrypt an item by selecting it and clicking the Encrypt button. The Encrypt *Class* dialog box appears, as shown in Figure 333.

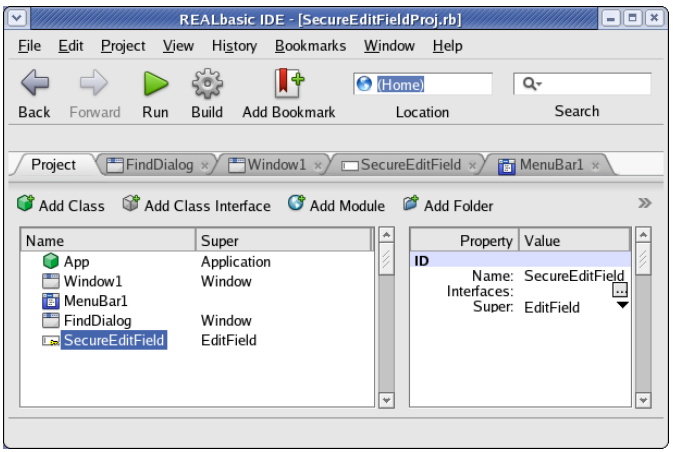
Figure 333. The Encrypt *Class* dialog box.

2 Enter and confirm a password for encryption.

Important: Don't forget your password.

An encrypted class appears in the Project Editor with a small key in the right corner of the class icon. If it is subclassed from a control class, it has the icon of the control.

Figure 334. A project with an encrypted class.



When a programmer tries to open an encrypted class, REALbasic presents the Decrypt *class* dialog box, shown in Figure 335.

To decrypt an encrypted class, do this:

1 Click on the module to be decrypted in the Project Editor and choose Edit ▶ Decrypt or Control-Click on the module in the Project Editor and choose Decrypt from the contextual menu or choose Edit ▶ Decrypt.

The Decrypt *class* dialog box appears.

Figure 335. The Decrypt *Class* dialog box.

2 Enter the decryption password and click Decrypt.

In a few moments, the key will disappear from the class's icon, indicating that it has been successfully decrypted. If you entered an incorrect password, a dialog box will inform you of that fact.

NOTE: Since other users cannot open an encrypted class to view its source code, you will need to provide them with a list of methods and properties if they should have access to them.

Deleting Classes From a Project

Before deleting classes from a Project, make sure you are not using the class in your code anymore. Also be sure to check for other classes that may have this class as their super class.

To delete a class from a Project, do this:

1. **Click on the class in the Project Editor to select it.**
2. **Press the Delete key on the keyboard or choose Edit ▶ Clear or Right+click (Windows and Linux) or Control-click on Macintosh on the class and choose Delete from the contextual menu.**

If you delete a class accidently, choose Edit ▶ Undo (Ctrl+Z or ⌘-Z).

CHAPTER 10

Creating Databases with REALbasic

With REALbasic, you can create database front-end applications that can be used with a variety of database engines, including REAL Software's own data source.

REAL Software's own database engine, REAL SQL Database, is built into REALbasic; both the Standard and Professional versions of REALbasic fully support this database engine.

Other database engines are supported only in the Professional version of REALbasic.

Contents

- REALbasic's database architecture,
- Structured Query Language (SQL),
- REALbasic's database tools,
- Creating and modifying databases from the Project Editor,
- The DatabaseQuery control,
- Using the DataControl control,
- Using object binding,
- Creating a database front end programmatically.

REALbasic's Database Architecture

You can use REALbasic to build a "front-end" to your database. It works in conjunction with a database "back-end" that actually stores the data itself. The database back-end can be a separate application or it can be REALbasic's own database back-end, REAL SQL Database. The front-end serves as the user interface — the means by which queries are sent to the source and information is displayed and printed. The end user uses the front-end to view, enter, and modify records, search for and sort records, and print reports.

The database back-end, such as REAL SQL Database, mySQL, PostgreSQL, 4D's 4D Server, or Oracle, actually stores the data. The database application that actually holds the data is referred to in REALbasic as the "data source."

A great feature of this architecture is that a database front-end that you create in REALbasic can be adapted to work with any supported data source—or multiple data sources. You can develop a database application with the internal REAL SQL Database and then deploy the system after switching the data source.

A REALbasic front-end can also use two or more data sources simultaneously. For example, you can access data locally on REAL SQL Database while simultaneously accessing remote data on a SQL or ODBC-compliant database.

REALbasic uses its plug-in architecture to support multiple data sources. Plug-ins are external files that must be placed in the Plugins folder in order to use the data source. The exception is the REAL SQL Database, which is built into the REALbasic application and does not require an external plug-in.

You (or a third-party) can add support for additional data sources by writing a plug-in for that back end. REAL Software's plug-in SDK contains information on writing database plug-ins.

Structured Query Language

A REALbasic front-end uses the Structured Query Language (SQL) to communicate with its data sources. The plug-in for your data source receives a SQL statement from REALbasic and (if necessary) translates the statement into a form that the data source understands, and sends it to the data source.

The REAL SQL Database is based on SQLite. It supports the subset of SQL described at http://www.sqlite.org. The 4D Server data source end is supported by the subset of SQL that versions of the REALdatabase up to version 5.0 used. The remaining data sources are native SQL back-ends. When you are working with any native SQL back-end, REALbasic simply passes your SQL to the data source. Therefore any valid SQL for that data source will work in a REALbasic front-end for that source. Please refer to the documentation for your selected data source for further information on supported SQL and specifics on the data types supported by that data source.

If you are unfamiliar with SQL, you will need to learn its basics before implementing your REALbasic front-end. This manual does not attempt to teach you SQL; rather, it describes the subset of SQL that is currently supported for the REAL SQL Database data source. For complete information on SQLite, see their web site at http://www.sqlite.org.

For other SQL data sources, please consult one of the many good SQL references, such as *SQL for Dummies* by Allen G. Taylor (ISBN: 0-7645-0105-4), *The Practical SQL Handbook*, by Bowman, Emerson, and Darnovsky (ISBN: 0-2014-4787-8).

REALbasic's Database Tools

A database front-end typically uses a mixture of database-specific and generic controls and commands. There are two database-specific controls, the DatabaseQuery and DataControl controls, and several classes and methods that are database-specific. Beyond that, you will use generic controls such as StaticTexts, EditFields, PopupMenus, ComboBoxes, and ListBoxes to display and edit data, PushButtons and menu items to perform actions, and TabPanel controls and other interface elements to polish the user interface.

All REAL SQL Database tables have an Integer Primary Key column. If you don't explicitly define one, one will be created for you. You can refer to the Integer Primary Key column using the "rowid" keyword. But, if you don't explicitly define your own Integer Primary Key column, you won't get the "rowid" column unless you refer to it in queries.

The REAL SQL Database supports text encodings. When you insert records into the database, the database stores text values using UTF-8. If it is not already UTF-8 it will be converted. If you want to defeat this conversion, store text data in a Blob column.

The REAL SQL Database engine supports transactions for both changes to the database design and for changes to the data. A transaction is started automatically when you make any change to the database and ended by a call to the SQL commands COMMIT TRANSACTION or ROLLBACK TRANSACTION.

Selecting a REAL Data Source

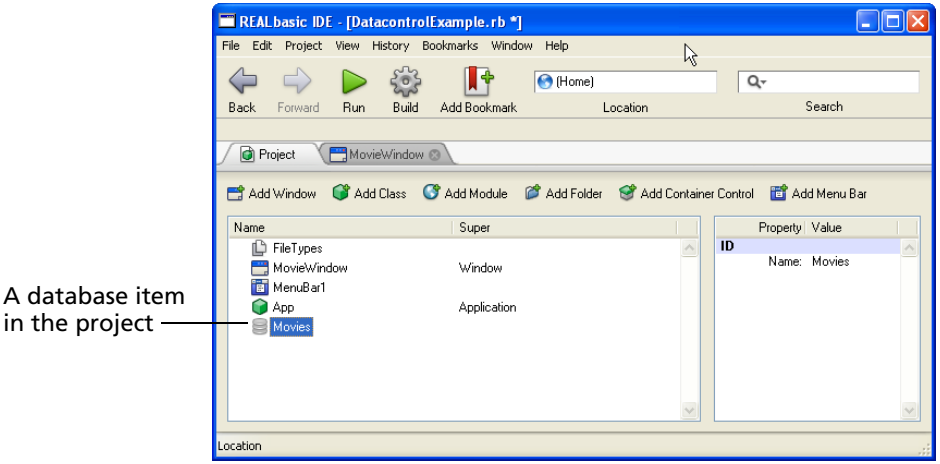
You can select a data source using either the language or the Project ▶ Add ▶ New REAL SQL Database or Select REAL SQL Database submenus in the IDE. The latter provides support for the REAL SQL Database engine plus any optional data sources for which you have installed plug-ins. Except for the REAL SQL Database, a data source will appear in this menu only if its plug-in is in the Plugins folder.

When you choose Select REAL SQL Database, a standard open-file dialog box appears. Navigate to the directory that contains the REAL SQL Database and open it.

When you choose New REAL SQL Database, a standard save-file dialog box appears. You can save the REAL SQL Database on any mounted volume. Name the database file as you would any other file and click Save.

When you choose an existing data source or add a new REAL SQL Database, it appears in the Project Editor as shown in Figure 336.

Figure 336. A database in the Project Editor.



To remove a data source from the Project Editor, highlight the data source and press the Delete key on the keyboard or choose Edit ▶ Delete. You can also Right+click (Control-click on Macintosh) on the data source and choose Delete from the contextual menu.

You can also create or open a REAL SQL Database using the language. To open an existing REAL SQL Database, create an instance of the REALSQLdatabase class and assign the location to its DatabaseFile property. Then call the Connect method. If Connect returns True, the connection was successful and you can proceed with database operations. Otherwise, you should look at the ErrorMessage and ErrorCode properties to troubleshoot the problem.

Here is an example:

```
Dim dbFile as FolderItem
Dim db as REALSQLdatabase
db=New REALSQLdatabase
dbFile = GetFolderItem("Pubs")
db.DatabaseFile=dbFile
If db.Connect() then
 //proceed with database operations here..
else
 Beep
 MsgBox "Error message: "+db.ErrorMessage
end if
```

This example opens the “Pubs” database, which is in the same folder as REALbasic. If the database is not in the same directory as the REALbasic application, use the

Volume function and the Parent or Child properties of the FolderItem class to navigate to it. Examples of this are shown in the section "Getting a File at a Specific Location" on page 382.

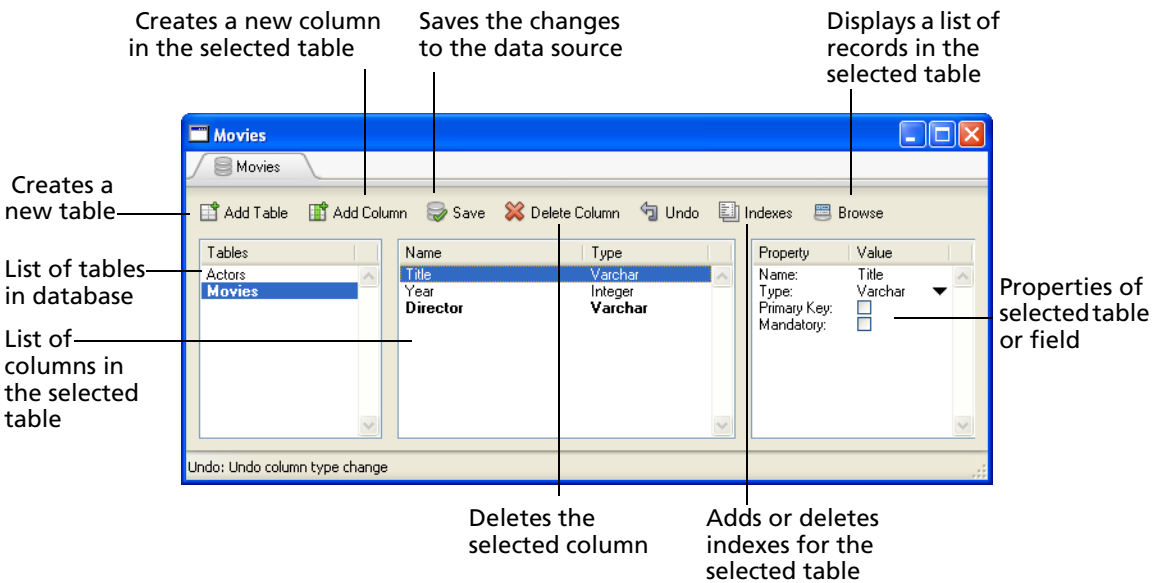
To create a new REAL SQL Database, follow the same procedure except call CreateDatabaseFile instead of Connect. It returns True if the operation was successful. Here is an example:

```
Dim db as REALSQLdatabase
Dim f as FolderItem
f=New FolderItem("mydb")
db=New REALSQLdatabase
db.databaseFile=f
If db.CreateDatabaseFile then
 //proceed with database operations...
else
 MsgBox "Error message: "+db.ErrorMessage
end if
```

Creating and Modifying Databases from the Project Editor

You can double-click a REAL SQL Database in the Project Editor to display its Schema—the list of its tables and each table's columns. From that list, you can view the data itself and the list of columns and their properties.

Figure 337. A database Schema for an existing database.



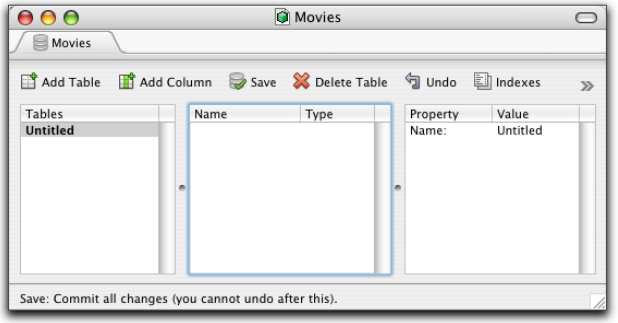
If the database is new, the list of tables will be blank; you can add tables by clicking the Add Table button.

To create a new table, do this:

1 **Click the Add Table button.**

REALbasic adds a new table to the database named "Untitled". The Properties pane shows that it has one property, its Name.

Figure 338. A new table in the Database editor.

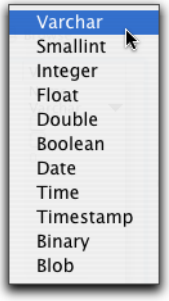


2 **Enter the Name of the table into the Properties pane.**

When you do so, the name of the table changes in the Tables pane.

Each table has to have at least one column. This is a unique identifier (Integer) that you can use to identify each record in relational operations. You don't need to create a Primary Key column of your own, but it is convenient to do so. If you refer to the "rowID" column, it will created for you.

3 **Click the Add Column button in the Table editor toolbar to add the first column.**



REALbasic adds a column named "Untitled" in the Columns pane, with the VarChar data type.

4 **Select the new column and use its Properties pane to set its Name and data type properties.**

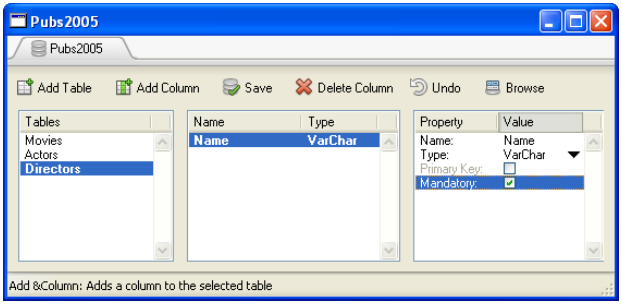
The supported column types are shown in the Type pop-up menu.

5 **Assign any other properties to the column, as desired. If the data source does not support a property that is listed in the Properties pane, it is dimmed out.**

6 **Repeat steps 3 and 4 to add additional columns.**

As you add each column, it appears in the column list. Figure 339 shows the Table editor with a new table and one column.

Figure 339. The New Table dialog with one column.



7 When you are finished specifying the table, click the Save button in the toolbar to commit all the changes to the data source.

Adding Indexes

After adding your columns, you will want to index certain columns. Indexes improve the performance of the database. You should index columns on which you plan to do the majority of your searches and sorts. Information retrieval and relational operations among tables are faster when indexed columns are used.

Note Certain data sources do not support indexing. If the data source you are working with does not support indexing, the Indexes button described in this section does not appear in the New Table/Edit Table dialog

Columns that are not good candidates for indexing are those that only take on a few values (i.e., gender or race), columns that are rarely searched on, and any columns in small tables where search and sort times are unlikely to be long.

You should not index too many columns because each index adds to the size of the database and, as records are added and deleted, it takes time for the system to update each index.

You can create indexes in your code via the SQL Create Index statement. If you want to create an index within the REALbasic IDE, use the following procedure.

To create the indexes for a table, do this:

1 Click the Indexes button in the Database editor toolbar.

REALbasic adds an Indexes panel to the editor.

2 Click Add Index to add a new index to the list of indexes.

A new untitled index appears in the Indexes list. You define this index by adding columns to it.

3 Rename the index to something more meaningful.

Usually you will use the names of the column or columns that comprise the index.

4 Click the Add Column button in the Indexes toolbar.

The Select Column dialog box appears, listing all the eligible columns in the selected table.

5 **Click on a column name to add it to the index.**

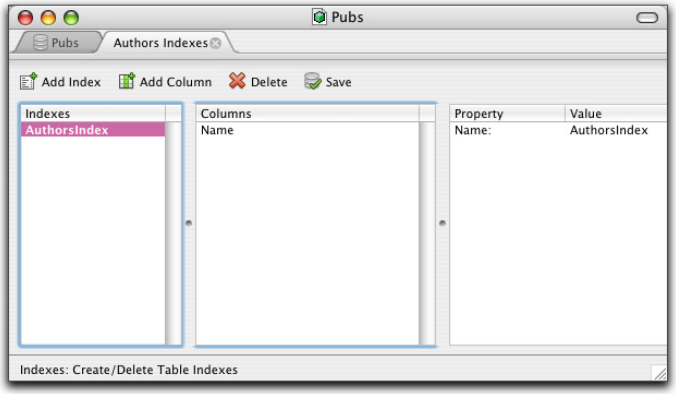
REALbasic adds the column name to the list of columns in the center of the Indexes editor.

6 **(Optional) To build a composite index (an index on two or more columns), click on Add Column once again and add another column to the index.**

For example, you may want to create a composite index on LastName and FirstName columns so that the database can quickly work with different people with the same last name.

Creating a composite index is different from creating several distinct indexes (such as the Primary Key index and the Name index). You create several distinct indexes by clicking the Add Index button once per index and adding one column per index. A finished index is shown in Figure 340.

Figure 340. The index for the Authors Name column.



7 **(Optional) To create additional indexes, click Add Index once again and repeat the process.**

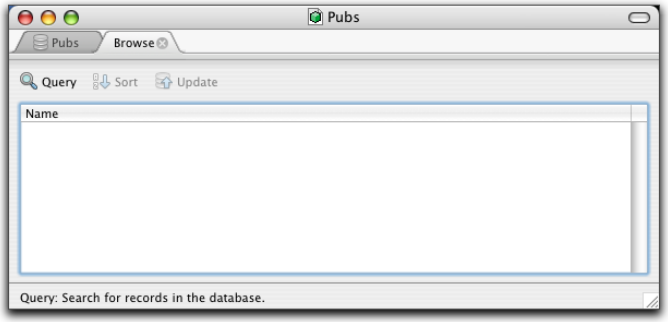
8 **Click Save in the Indexes toolbar to save the indexes to the database.**

Viewing Data

The Database Editor window enables you to view, sort, and update records, but it isn't a full-fledged end-user interface. You use a SQL Select statement to retrieve and sort the records. If the Select statement has no errors, REALbasic will display the requested records in a list. If it encounters an error, it will display an error code and message. REAL SQL Database returns SQLite error codes. See the entry in the Language Reference for the REALSQLDatabase class for information those error codes or http://www.sqlite.org.

To view the existing data, click the Browse button in the Browse toolbar. REALbasic adds a new tab panel to the window for listing records. This is shown in Figure 341.

Figure 341. The Browse panel before running a query.

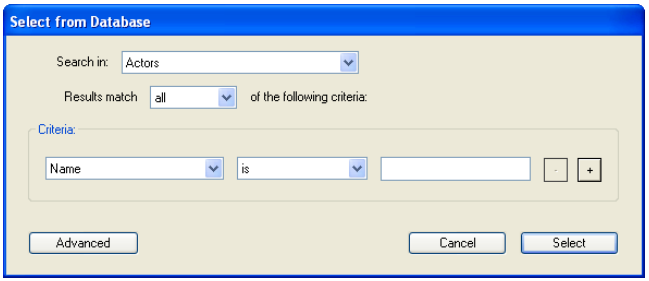


Initially, no records are shown. You must first select records to view via a Query.

Click the Query button in the Browse panel to display the Query dialog box. It enables you to specify a query via a "Find" dialog, shown in Figure 342.

The Select From dialog box builds a SQL Select statement for you from your specifications. You can also do the query by writing a SQL Select statement manually. If you wish to specify the query that way, click the Advanced button. It is described next, in the section "Advanced Select" on page 485.

Figure 342. The Select From Query dialog box (Simple screen).



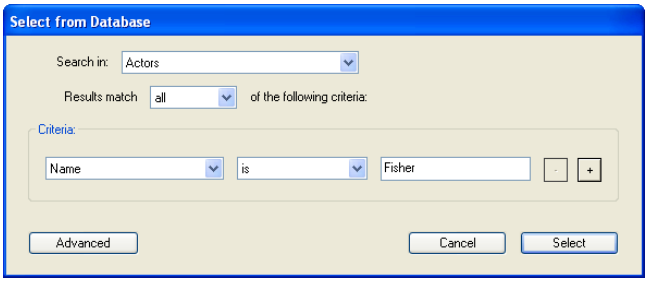
The simple version of the dialog box enables you to specify your query without knowing about the syntax of a SQL Select statement. The elements of this dialog perform the following functions:

- The **Search In** drop-down list contains the list of all of the tables in the database. You can search in only one table. First choose the table that will be searched. Figure 342 specifies that this query will be in the Actors table.
- The **Criteria area** is where you specify the query. Each row in the Criteria area allows you to query on one column. You can specify multiple queries and combine them with either the AND or the OR operators. The first drop-down list in the Criteria area lists all the columns in the selected table. Choose a column to search on. Figure 342, for example, specifies the Name column in the Actors table.
- The **Value area** is where you specify the value you are looking for in the search.

- The **Operators drop-down list** lets you specify the relationship of the column's contents to the value you entered in the Value area. Your choices are "is" (a.k.a. equal to), "is not" (a.k.a. not equal to), "greater than", "less than", "starts with", "ends with", and "contains". Greater than and less than pertain to numeric, date, and time columns, while starts with, ends with, and contains pertain to VarChar columns.

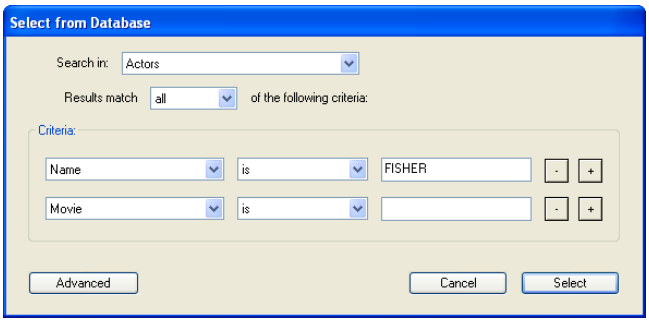
  For example, the following dialog specifies "Name equal to Fisher".

Figure 343. A completed simple search.



If you want to search on more than one column, you need to add a row to the Criteria area. Do this by clicking the Plus sign (to the right of the Value area). A new blank row is created in the Criteria area. This is shown in Figure 344.

Figure 344. A second criterion row in the Criteria area.



Repeat the process of specifying the search criterion on the second column.

If you want to remove one of the rows in a multiple-column search, click the minus sign (next to the plus sign) in the row you want to remove.

When you have two or more rows in the Criteria area, you must specify whether a record must meet all of the specifications or any one of the specifications. The first type of search uses the AND operator to relate all statements, while the second type of search uses the OR operator.

You do this using the "Results Match" drop-down list. The default selection, "All", requires that a record is not selected unless all of the statements in the Criteria area

are true; the "Any" selection means that a record will be selected if any one of the statements in the Criteria area are true.

Figure 345. The Results Match drop-down list.

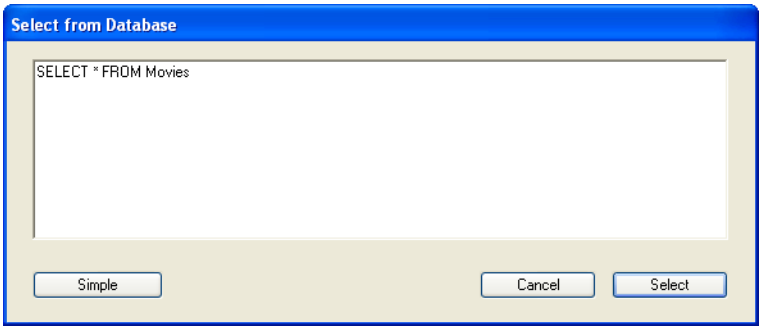
If you search on only one column, the Results Match drop-down list is not relevant.

When you are finished specifying the query, you can click Select to do the search.

Advanced Select

Figure 346. If you want to enter the SQL Select statement directly or modify the one that is generated from the "Simple" version of the dialog box, click the Advanced button. The Advanced version of the dialog has only one field, in which you must write a valid SQL Select statement.

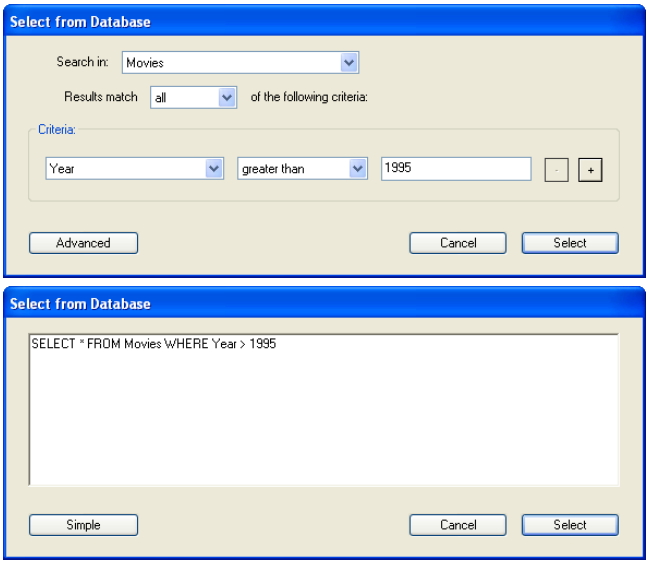
Figure 347. The Advanced version of the Select From dialog box.



If you have already entered search specifications into the "Simple" version of the dialog, those specifications will have been translated into a valid SQL Select statement for you. You can then edit that query or replace it.

For example, here is the same query that expressed on both dialog types:

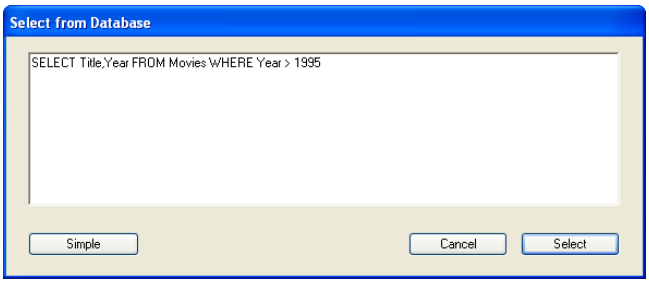
Figure 348. A query in the Advanced and Simple versions of the dialog box.



You may, of course, continue editing the SQL statement that REALbasic generated. The Simple search screen does not contain interface elements for all of the options that the SQL Select statement supports, so you may want to take advantage of these options. For example, the Simple screen does not enable you to specify the list of columns that will be returned by the query. Any search that is specified on the Simple screen will return all the columns.

If you want fewer columns, you should replace the asterisk in the SQL statement with the column list. For example, Figure 349 shows an edited version of the query in Figure 348 that specifies two columns.

Figure 349. A modified SQL query.



If you switch to the Advanced version of the dialog after entering a SQL query, REALbasic will set the values of the drop-down lists and the Value area to the extent that it can. If you have specified options in the SQL statement that are not available in the Advanced view, you will lose that information.

In the Simple view, you can add an ORDER BY clause to sort the rows by one or more columns. Figure 350 shows a SQL statement that uses the ORDER BY clause to sort all the movies in the table in descending order.

Figure 350. A listing of all records.

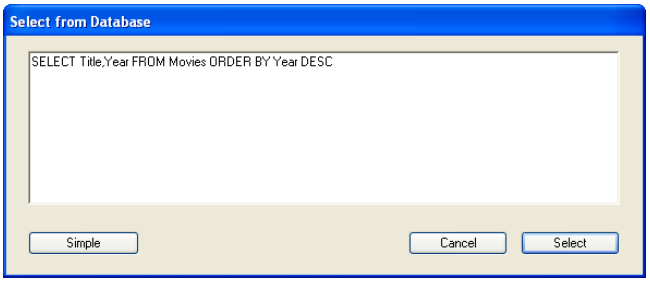
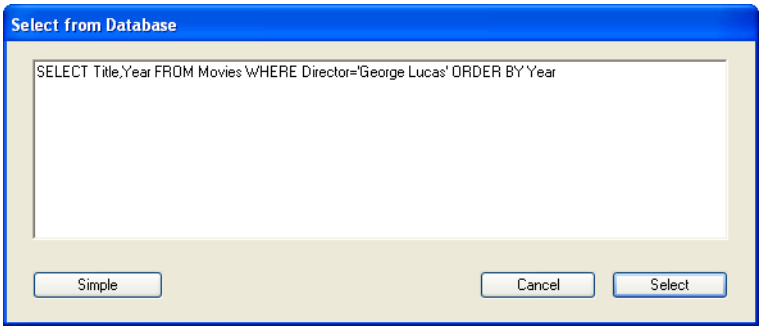


Figure 351 shows another example of a SQL statement that returns two columns and the records sorted in ascending order (the default sort order).

Figure 351. Selecting and sorting rows with a SQL Select statement.



Storage Types and Column Type Affinities

The data types shown in Table 36 are supported by the REAL SQL Database. Except for the Integer primary key column, any column can store data of any type.

If you are using another data source, please consult the documentation for your data source for information about its supported data types.

Table 36: Data Types supported by REAL SQL Database

Data Type	Description
Binary	Stores code, images, and hexadecimal data of any size.
Blob	Stores a binary object. The REALSQLdatabase supports blobs of up to any size. A blob can be stored in a column of any declared data type.
Boolean	Stores the values of TRUE or FALSE.
Date	Stores year, month, and day values of a date in the format YYYY-MM-DD. The year value is four digits; the month and day values are two digits.
Double	Stores double-precision floating-point numbers.

Data Type	Description
Float	Stores floating-point numeric values with a precision that you specify, i.e., FLOAT (5).
Integer	A numeric data type with no fractional part. The maximum number of digits is implementation-specific. The REAL database supports 4-byte integers, which provide a range of ±2,000,000,000. If you are using another data source, check the documentation of your data source.
SmallInt	A numeric data type with no fractional part. The maximum number of digits is implementation-specific, but is usually less than or equal to INTEGER. The REAL database supports 2-byte smallints, which allow you to store values in the range of ±32,767. If you are using another data source, check the documentation of your data source.
Time	Stores hour, minute, and second values of a time in the format HH:MM:SS. The hours and minutes are two digits. The seconds values is also two digits, may include a optional fractional part, e.g., 09:55:25.248. The default length of the fractional part is zero.
TimeStamp	Stores both date and time information in the format YYYY-MM-DD HH:MM:SS. The lengths of the components of a TimeStamp are the same as for Time and Date, except that the default length of the fractional part of the time component is six digits rather than zero. If a TimeStamp values has no fractional component, then its length is 19 digits If it has a fractional component, its length is 20 digits, plus the length of the fractional component.
VarChar	Stores alphabetic data, in which the number of characters vary from record to record. The REAL SQL Database uses UTF-8 text encoding. If the text is not already in UTF-8, it is converted. If you want to preserve a different encoding, use a Blob column instead.

Values that are passed in single or double quotes (as literals) are stored as Text.

The DatabaseQuery Control

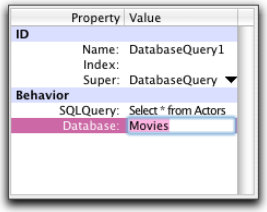
The DatabaseQuery control can be used to send queries to the data source, but this function can also be performed with the language. It is up to you.

You add a DatabaseQuery control to a window like any other control, but it is not visible to the end-user. It is used only as an object that performs database queries. It has the following properties:

Name	Description
Database	The data source that will be queried.
SQLQuery	The text of the SQL query to be run against *Database*

The SQLQuery that you enter in the Properties pane is executed automatically when its window appears. For example, the properties shown in Figure 352 will retrieve all rows and columns from the Actors table when the window opens. However, the DatabaseQuery control cannot *display* the rows and columns all by itself.

Figure 352. A DatabaseQuery control's Behavior properties.



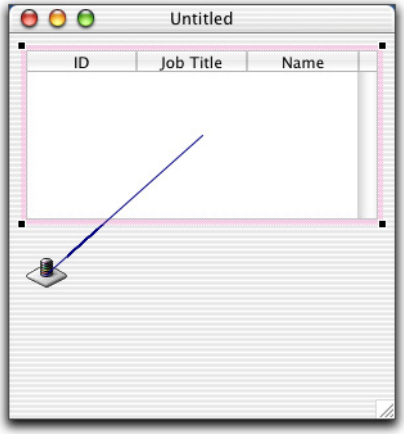
The DatabaseQuery control has one method, RunQuery, which executes *SQLQuery* against *Database*. You can call it via your code.

Using Object Binding with a Database Query

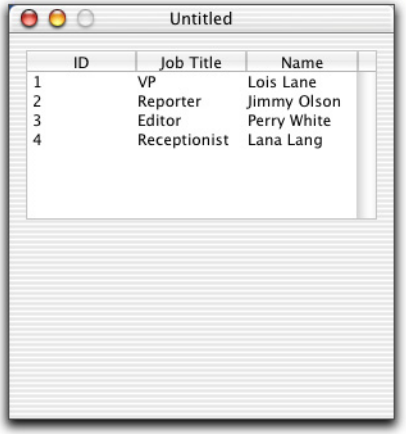
The DatabaseQuery control, together with object binding, allows you to create a simple database front-end with no programming. Since the DatabaseQuery control automatically executes *SQLQuery*, you only need a means to display the results. Simply add a ListBox to the window and bind the DatabaseQuery and the ListBox controls (select both the DatabaseQuery control and the ListBox and choose Project ▶ Add ▶ Binding). An Object Binding dialog box will appear. Select the option that binds the ListBox to the DatabaseQuery control's results.

When you click the Run in the Main Toolbar, the data appear in the ListBox, as shown in Figure 353.[1]

Figure 353. A Simple database built with no programming.



Design Environment



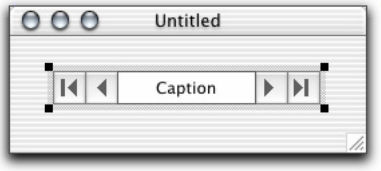
Runtime Environment

1. Headings were added to the ListBox using its Properties pane.

The DataControl Control

The DataControl control gives you a very simple and powerful way to create a data entry screen that works with a database table. The DataControl is a single object that consists of record navigation buttons (First, Last, Next, and Previous records) and a caption. When you place it in a window, it looks like this:

Figure 354. A DataControl in a window.



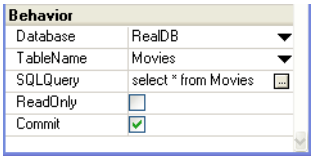
You use a DataControl by creating a window that uses EditFields, ListBoxes, Popup Menus, ComboBoxes, or StaticText controls to display and edit data. Custom classes based on these controls can also use a DataControl as their data source.

These controls have two properties that are meaningful only when used in conjunction with a DataControl: DataSource and DataField. They are shown in the Database Binding topic at the bottom of the control's Properties pane:

Figure 355. The Database Binding properties of an EditField.

The DataSource property should be assigned the name of the DataControl in the window. It, in turn, is bound to a database table using its Database and TableName properties. When you set the Database, a pop-up menu of tables from the database becomes available for the TableName property.

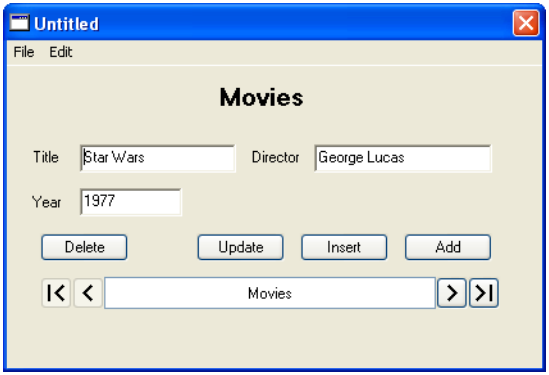
Figure 356. The DataControl's Behavior properties.



These properties specify that the **DataControl** will manage the records in the Movies table in the database "RealDB." The SQLQuery property is the SQL query that will run when the form is opened. This SQL statement finds all records in the Movies table.

The following window displays columns in EditFields and uses a DataControl for record navigation.

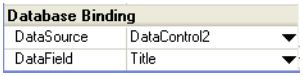
Figure 357. A simple database that uses a DataControl.



Each EditField is linked to the **DataControl** in its Properties pane by setting the DataSource and DataField properties.

The EditField for the movie title is bound to the column "Title" in the table to which the **DataControl** is linked.

Figure 358. The Database Binding properties for the "Title" EditField.



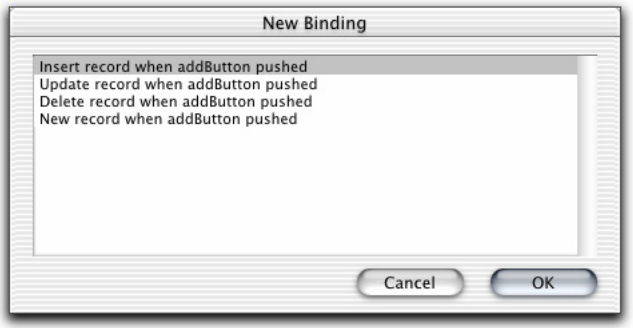
Each field on the form is bound to a column in the Movies table in this manner.

The DataControl also enables you to program buttons, CheckBoxes, and RadioButtons using Object Binding. The easiest way to set up controls for creating, modifying, or deleting records is to bind them to the DataControl. The following actions can be assigned to controls via Object Binding:

- The Insert bind inserts the record currently shown in the window into the table,
- The Update bind saves changes to an existing record,
- The New bind clears the contents of the controls,
- The Delete bind deletes the record from the table.

To establish the bind, select both the PushButton control and the DataControl. You can do this by Ctrl+clicking on both controls (Command-clicking on Macintosh) or by drawing a selection rectangle around both controls. Next, choose

Project ▶ Add ▶ Binding to display the New Binding dialog box and select "New Record when addButton pushed".



You may have added Binding to your Window Editor toolbar. If so, you can click on it instead of choosing Project ▶ Add ▶ Binding.

Use the same process to bind the Delete button to the **DataControl** and use "Delete Record when DeleteButton pushed." Bind the Insert button to the "Insert record when InsertButton pushed" and bind the Update button to the "Update record when UpdateButton pushed.

To add a new record, click the Add button, enter the information into the fields, and click Insert. To modify a record, navigate to it using the buttons in the DataControl, edit the values, and click Update. To delete a record, navigate to it using the buttons in the DataControl and click Delete.

These possible actions supported by Object Binding for a DataControl are summarized in Table 37.

Table 37: Object Binds for the DataControl control.

Source Control	Available Object Binds
PushButton	Insert Record when PushButton is pushed. Update Record when PushButton is pushed. Delete Record when PushButton is pushed. New Record when PushButton is pushed.
BevelButton	Insert Record when BevelButton is pressed. Update Record when BevelButton is pressed. Delete Record when BevelButton is pressed. New Record when BevelButton is pressed.
RadioButton	Insert Record when RadioButton is selected. Update Record when RadioButton is selected. Delete Record when RadioButton is selected. New Record when RadioButton is selected. Insert Record when RadioButton is deselected. Update Record when RadioButton is deselected. Delete Record when RadioButton is deselected. New Record when RadioButton is deselected.

Table 37: Object Binds for the DataControl control.

Source Control	Available Object Binds
CheckBox	Insert Record when CheckBox is checked. Update Record when CheckBox is checked. Delete Record when CheckBox is checked. New Record when CheckBox is checked. Insert Record when CheckBox is unchecked. Update Record when CheckBox is unchecked. Delete Record when CheckBox is unchecked. New Record when CheckBox is unchecked. Enable DataControl when CheckBox is checked.

With object binding, you can set up button actions for these three key operations without coding.

Creating a Database Front End Programmatically

To fully exploit REALbasic's database capabilities, you will need to write some code. The commands that are listed in the Database theme in the *Language Reference* provide you with all the necessary tools to build sophisticated database front-ends.

Choosing a Data Source

The following classes allow you to choose the database back-end(s) used in your application. Except for the REAL SQL Database, each class requires that the appropriate database plug-in be installed in REALbasic's Plugins folder. All the database plug-ins are available from REAL Software. Database plug-ins may be updated more frequently than REALbasic itself; the latest versions of each plug-in can be found at www.realsoftware.com.

FrontBase can also be accessed from REALbasic using a third-party plug-in. The FrontBase plug-in is free and is available on the from the REALbasic web site.

Data sources are accessed via a group of subclasses of the Database class. Each subclass accesses a specific data source. The database subclasses shipped with REALbasic are shown in the following table. Third parties may also write plug-ins that support other data sources.

Table 38: Classes used to access data sources.

Database Subclass	Description
MySQLDatabase	Accesses MySQL databases.
ODBCDatabase	Accesses ODBC-based databases.
OpenBaseDatabase	Accesses an OpenBaseDatabase.
OracleDatabase	Assesses Oracle 8i and above databases.
PostgreSQLDatabase	Accesses a PostgreSQLDatabase.
REALSQLdatabase	Built into REALbasic and supports the REAL SQL Database. The REAL SQL Database data source is supported in both the Standard and Professional version of REALbasic.

Table 38: Classes used to access data sources. (Continued)

Database Subclass	Description
Database4DServer	Supports 4D Server versions 6.8 and above, including 4D 2003-2004. Access to 4D Server via ADSP is not supported.

Please see the entries for the new Database subclasses in the *Language Reference* for more information.

When opening a data source, you should test whether your connection is successful before proceeding with database operations. Use the Connect method of the Database class to determine whether the connection was successful. If it was, Connect will return a value of True. If the connection failed, you should examine the contents of the ErrorMessage and ErrorCode properties of the Database class.

For example, the following code opens a mySQL database.

```
Dim db as mySQLDatabase
db=New mySQLDatabase
db.host="192.168.1.172"
db.port=3306
db.databaseName="MyOwnmySQLdatabase"
db.userName="Mary"
db.Password="Elton"
If db.Connect then
  //proceed with database operations
else
 MsgBox "Connection failed!"
end if
```

With the exception of OpenCSVCursor, these methods return an object of type Database. Table 39 gives the Database Class methods.

Table 39: Database Class methods.

Name	Parameters	Description
Close		Closes the database.
Commit		Commits (saves) changes to records. If you quit the application after making changes to a RecordSet, REALbasic issues an implicit Commit. Use Commit and Rollback to manage transactions.
Connect		Connects to the database server and opens the database for access. Returns a Boolean. Before proceeding with database operations, do a test to be sure that Connect returns True.
FieldSchema	TableName as String	Returns a RecordSet with information about all columns in the table. See notes, below. In versions 4.0 and earlier, it returned a DatabaseCursor.

Table 39: Database Class methods. (Continued)

Name	Parameters	Description
InsertRecord	TableName as String, Data as DatabaseRecord	Inserts *Data* as the last row of *TableName*.
Property	Name as String	Returns the database property specified by *Name* from the data source. Currently supported only for 4D Server and for getting the connection string.
Rollback		Cancels a set of changes to records.
SQLSelect	SelectString as String	The SQL Select statement. Returns a RecordSet. In versions 4.0 and earlier, it returned a DatabaseCursor.
SQLExecute	ExecuteString as String	The SQL Execute statement. Used to execute SQL commands that do not return a RecordSet.

The SQLSelect method can be used instead of a DatabaseQuery control to send queries to the database. The SQLExecute method can be used in place of the database Schema dialog boxes to build database schema as well as perform many other functions. For example, the following statement can be used to generate a table in place of the interactive method discussed in the section, "Creating and Modifying Databases from the Project Editor" on page 479.

```
Dim db as Database
.
.
db.SQLExecute("create table authors(au_id integer not null, au_fname
varchar, au_lname varchar, address varchar, city varchar, state varchar, zip
varchar, phone varchar, primary key (au_id)")
```

Editing Records

The SQLSelect method returns an object of type RecordSet. In the terminology of SQL, a RecordSet is a database *cursor*. You may encounter the term cursor instead of recordset when you consult SQL reference material.

The RecordSet contains the records that meet the selection criteria (defined by the SELECT statement's WHERE clause) and, in multi-user applications, locks them against modification by other users. You can then display the rows and/or edit them. If you need to edit the rows, you must process the records one row at a time. Please refer to the entry for RecordSet in the *Language Reference* for a description of its properties and methods.

When an SQL statement returns a RecordSet, the user has 'possession' of those records for his exclusive use. If the RecordSet contains more than one record, you can use the RecordSet's properties and methods to cycle through the rows and columns of the RecordSet.

After a query using the SQLSelect method, the RecordSet contains a pointer to the current record (by default, the first record in the RecordSet). You can use the BOF (Beginning Of File) and EOF (End Of File) properties of the RecordSet class to

determine if the current record pointer is before the first record or beyond the last record. You can use these to determine if your query has found any records. If your query finds no records, both BOF and EOF will both be True since the current record pointer points at nothing.

If your query finds one or more records, both BOF and EOF will be False, since the current record pointer is pointing at the first record and not at the beginning or end of the file. For example, if your query found three records, the RecordSet would look like this:

BOF
Record1 (the current record pointer is pointing at this record)
Record2
Record3
EOF

As the record pointer moves to a particular record, you can use the Edit and Update methods to edit the record or the DeleteRecord method to remove the record. To modify the record, call the Edit method and perform the modifications.

To get or set the value of an individual field, use the Value property of the DatabaseField class. The Value property is a Variant, so you can use it to work with columns of any type. To set the value of a field to Nil, assign Null to the Value property for that field.

After modifying the columns for a row, call Update to update the RecordSet and when you are finished, call the Database object's Commit method to commit the set of modifications to the database, or call the Rollback method to cancel the modifications. The EOF property becomes True when you try to move the pointer past the last record.

If you don't call Commit, REALbasic issues an implicit Commit when the user quits the application.

See the examples for the RecordSet, DatabaseField, and Database classes in the *Language Reference* for more information.

Adding Records

You add a new record using the Database object's InsertRecord method. It has two parameters, the database object and an object of type DatabaseRecord.

Name	Type	Description
Column	Name as String	Column in current table.

For example, the following code adds a record to the "employees" table.

```
dim db as Database
db=New Database
.
dim rec as DatabaseRecord
rec = New DatabaseRecord
.
rec.Column("au_id") ="09"
rec.Column("au_fname") ="Oscar"
rec.Column("au_lname")="Wilde"
db.InsertRecord("authors",rec)
```

See the descriptions of the Database, RecordSet, and DatabaseRecord classes in the *Language Reference* for additional discussion and examples.

CHAPTER 11

Debugging Your Code

Wouldn't it be great if every line of code executes just the way you want without a single error? Well, for those times when it doesn't work out that way, REALbasic provides you with some tools to track down the bugs and fix them.

Contents

What is Debugging?

Debugging means removing errors, both logical and syntactical, from your programming code. Errors in programming code are referred to as "bugs." You are probably wondering why errors are called "bugs." Well, back in the 1940's, the United States Navy had a computer that occupied an entire warehouse. At that time, computers used vacuum tubes and the light from the tubes attracted moths. These moths would get inside the computer and short out the tubes. Technicians would have to go in and remove the bugs to make the computer work again. Since this was a government project, everything had to be logged, so they would put down "debugging computer" in the log. But enough of the history lesson.

Debugging is part of programming. It's the part of programming most programmers like the least. Fortunately, REALbasic makes it easy to track down those nasty bugs and squash them like a, well, bug. REALbasic comes with a Debugger which is a set of windows that help you see what is going wrong.

Logical Bugs

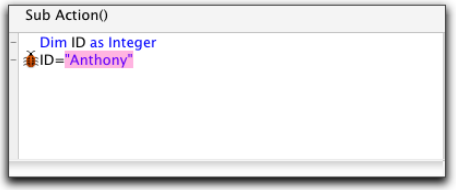
These are bugs in your programming logic. You will know you have found one of these when your code zigged when it should have zagged. REALbasic's built-in Debugger can help you find these by letting you watch your code execute one line at a time.

Syntactical Bugs

These are bugs where you have mistyped the name of a class, property, variable, or method. You may have also tried to use two values together that don't go together. For example, if you try to assign a string value to a variable or property of type integer, you will get a Type Mismatch error because they are different data types.

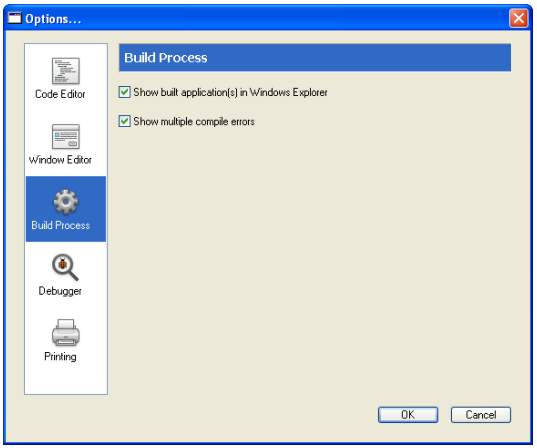
REALbasic makes finding syntax errors a snap. When you click the Run button in the Toolbar or choose Project ▶ Run (Ctrl+R on Windows and Linux or ⌘-R on Macintosh), REALbasic checks your code for syntax errors and reports them instead of compiling your project. It puts a bug icon to the left of each line containing the bug. Yes, you have to fix all of them before you can compile the project. Figure 359 shows an example of an error identified by REALbasic.

Figure 359. A bug in the Code Editor.



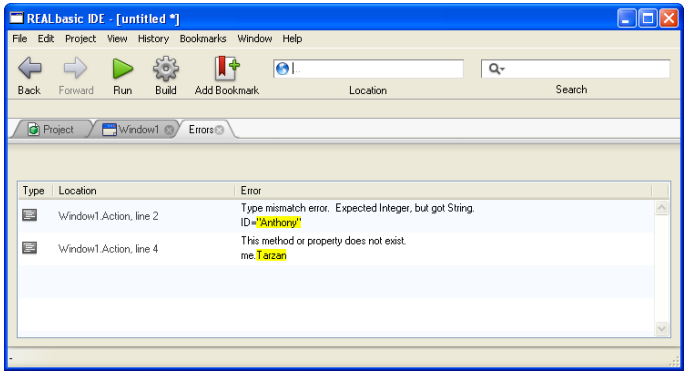
For large projects, it is often more efficient to get a list of all your syntax errors at once. To do this, use the "Show Multiple Compiler Errors" option in REALbasic Options. Choose Edit ▶ Options on Linux and Windows or REALbasic ▶ Preferences on Macintosh and select the "Show Multiple Compiler Errors" option on the Build Process options screen.

Figure 360. The Show Multiple Compiler Errors option in Options.



If this option is selected and you have more than one syntax error in the project, all the errors are listed in an Errors screen instead of being displayed as tips in the Code Editor. Figure 361 shows a Multiple Compiler Errors window after an attempted build.

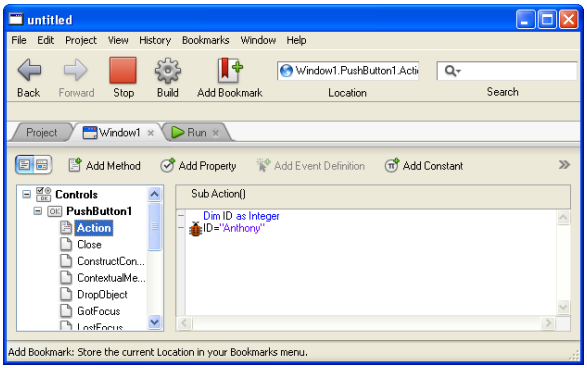
Figure 361. Multiple errors in the Errors screen.



Each entry in the Errors screen gives the type of error, the location of the error, and the line number that contains the error. In Figure 361, the errors are type mismatch errors. You can fix the error either by double-clicking on the line to go to the line in the Code Editor.

To go to the line of code directly, double-click on an item in the Errors screen. REALbasic will then open that Code Editor and you can edit the incorrect line there. The line that contains the error will have a bug to its left. For example, in Figure 362 the first error listed in Figure 361 is shown. The Code Editor shows PushButton1's Action event handler.

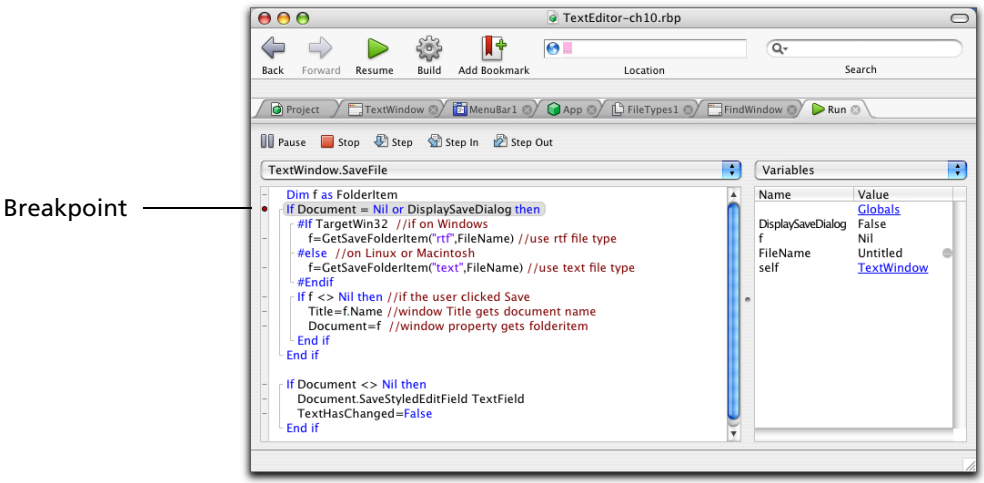
Figure 362. The first error in the Errors screen shown in Figure 361.



The Debugger

When there are no syntax errors, the test application launches and a Debugger screen called "Run" appears in the REALbasic IDE. It enables you to execute your code line-by-line while watching the values of your variables. The Debugger highlights the line of code that is about to be executed.

Figure 363. The Debugger screen.



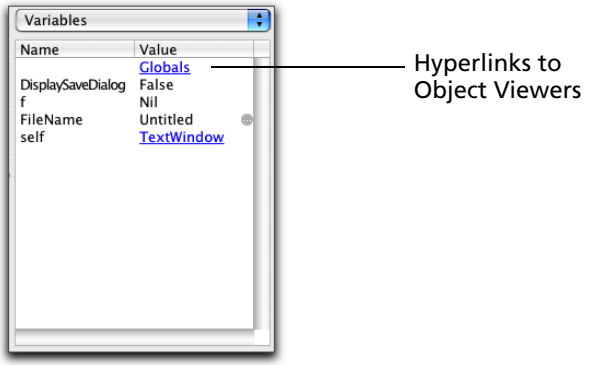
The Debugger screen is divided into two panes. The Code Editor section shows the method that is currently executing. Execution has stopped at the line of code that is highlighted. The red dot indicates the breakpoint.

The Stack drop-down list, just above the Code Editor, contains the name of the current method, along with the list of methods that eventually invoked the current method. They are listed in the order that they were called. You can check the Stack drop-down list to verify that methods are actually called when you expect them to

be called. You can view the code for any method listed in the Stack drop-down list by clicking on it.

The Variables pane contains a list of all the variables local to the method shown in the Code Editor area, along with their current values (if any).

Figure 364. The Variables pane.



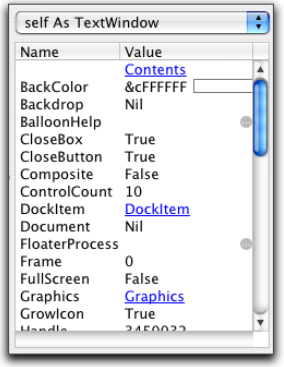
Any objects (rather than variables) that are defined in the method are shown as hyperlinks rather than values. If you click a link, the pane changes to show the current properties of the object you clicked. The top object, Globals, links to a pane that shows the values of your global variables, if any.

If you Shift-click (Command-click on Macintosh) on a hyperlink, the requested Object Viewer opens in a separate window. Otherwise, the new Object Viewer replaces the current Object Viewer in the IDE window.

You can also right+click (Control-click on Macintosh) on a hyperlink to display a contextual menu that lets you choose between opening the new Object Viewer in the IDE or in a new window. The latter is equivalent to Shift-clicking (Command-clicking on Macintosh) on the hyperlink.

Figure 365, for example, shows the Object Viewer for an application's main window. You get this by clicking the TextWindow link in Figure 364.

Figure 365. The Object Viewer for TextWindow.



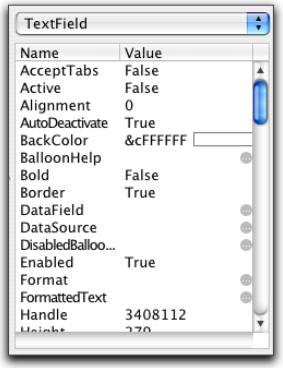
The important feature of the Object Viewers is that they are interactive. As you execute code line by line in the Debugger, the Object Viewer updates values in real time. In this way, you can see whether a particular value (or lack of a value) is causing the problem.

In Figure 365, the top link, Contents, displays a list of objects in the window. In this case, there are several buttons, a Canvas, and an EditField. Click the Contents link to examine these objects and click an object to display its variables.

Figure 366. The Contents of TextWindow pane.

Click a control to display its variables and objects, if any. In the case of TextWindow, several controls are contained within the window. They are listed on the Contents pane. For example, clicking the first link, TextField, shows its properties. As you drill down the hierarchy of objects, the pop-up menu above the variables pane enables you to jump back up to an earlier level in the hierarchy.

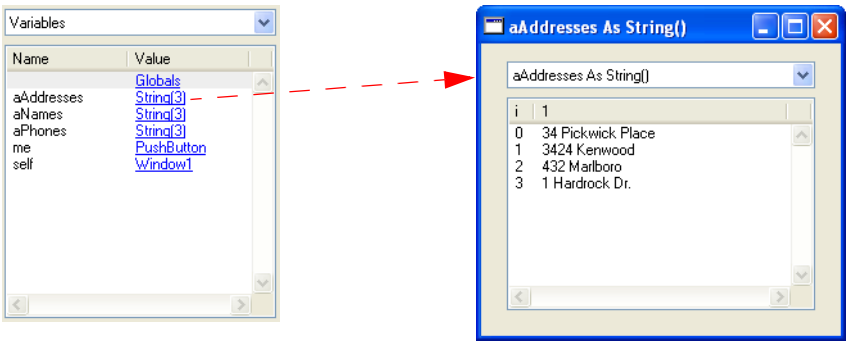
Figure 367. The Contents pane of TextWindow's Object Viewer and the Object Viewer of one of TextWindow's controls.



This sequence is repeated down to the last level of objects. For example, if a window contains an ImageWell control, it is listed on the window's Contents pane. If the ImageWell has a picture assigned to its Image property, the picture is shown on the Contents pane of the ImageWell's Object Viewer.

If the variable that you are investigating is an array, the array's Object Viewer displays each value in the array and its index. For example, the Variables Object Viewer below shows that there are three arrays. Right+clicking on the aAddresses array to display its Object Viewer shows the values of each of its elements.

Figure 368. A Variables Object Viewer and the Object Viewer for one of its arrays.



Controlling Execution

Using the Debugger's Toolbar, you can control execution. Instead of just allowing your code to run normally, you can control execution on a line-by-line basis.

The Debugger's Toolbar has five buttons that perform the functions of the Debug menu items:

Figure 369. The Debugger Toolbar.

When the Debugger is active the Project ▶ Step submenu offers the Step Over, Step In, and Step Out commands as well. In addition, the Run button in the Main Toolbar has changed to Resume.

- **Resume**: (Main toolbar) Continues execution from the breakpoint line without further interruption. This exits from the Debugger screen.
- **Pause:** Used to Pause execution without stopping and exiting to the IDE.
- **Stop**: Stops execution and returns to the REALbasic IDE. This also exits from the Debugger screen, but without executing any more code.
- **Step**: Executes the current line (indicated by the highlight) and moves on to the next line. If the current line includes one of your methods, the Debugger executes the method but will *not* step through the method's code.
- **Step In**: Executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger displays the method and steps through the method's code.
- **Step Out**: Executes the rest of the method without stopping on each line. This is handy when you have used Step In to step through a method that was called by another method and now wish to continue code execution without stopping on each line.
- **Edit Code:** Jumps to the Code Editor for the method that is currently executing. Use this button to modify code or correct errors in the current method. It does not close the Run screen.

There are several ways that REALbasic halts the compilation process:

You Have a Syntax Error In Your Code

When you attempt to compile your project, REALbasic checks your code for syntax errors. If it finds one, it stops immediately and either displays the Code Editor along with the error message as a Tip or it opens a new panel with a list of syntax errors. An example of this case is shown in Figure 359 on page 500.

You Have Set A Breakpoint In Your Code

A *breakpoint* is a marker you can set for any line of code that tells REALbasic to display the Debugger when it reaches that line of code but before it executes it. In the Code Editor, the lines that accept breakpoints are indicated by a dash in the first column, to the left of area that accepts code. You set a breakpoint simply by clicking on the dash.

In Figure 363 on page 502, a breakpoint is being set. A red dot will appear to the left of the line of code to indicate that a breakpoint has been set.

Breakpoints are persistent. This means they will stay in your code until you remove them. You remove a breakpoint by clicking on the red circle. You can also clear all breakpoints by choosing Project ▶ Clear All Breakpoints.

NOTE: Breakpoints have no effect in stand-alone applications you build by clicking the Build button in the Toolbar or by choosing Project ▶ Build Application.

You use the Break Keyword

Instead of setting a break point in the Code Editor as shown in Figure 363, you can place the Break keyword in your code. When the compiler reaches this line of code, it breaks into the debugger as if a breakpoint had been set. With the Break keyword, you can set the breakpoint conditionally. For example, you if you want to break into the debugger only when the value of a variable is equal to a certain value, you can put the Break keyword in an If statement that tests for that value. It's convenient to use the one-line version of the If statement, for example:

```
If ErrorNum=103 then Break
```

breaks into the debugger when a local variable equals 103.

You Have Pressed a Keyboard Equivalent

If you are running your project and you need to interrupt the code that is executing, press Control-C on Macintosh or either Ctrl+Break or F5 on Windows or Linux to halt code execution and display the Debugger. This is handy if you find yourself in an endless loop.

You can also switch back to the Development environment by clicking the Stop button in the Debugger Toolbar.

Command-Shift-Period does not work under Mac OS X.

Following the Execution of Methods

When your code isn't cooperating or you're just not sure what is executing and when, it's helpful to be able to watch your code as each line executes. The Debugger makes this easy. Once the Debugger is displayed, the current line (the line that s highlighted) indicates which line of code is about to be executed. When you tell the Debugger to continue, it executes that line and goes on to the next line of code. What it does next depends on the command you give it when you wish to continue. The Debugger Toolbar and the Project menu give you three commands, each of which will execute the current line and then take a different course of action for the next line of code.

Step

Step executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger executes the method but will *not* step through the method's code. When the method is finished executing, the Debugger

will continue from the next line of code in the current method. Consider the following code:

```
EditField1.SelBold=True
EditField1.Text=ToFrench(EditField1.Text)
EditField.SelBold=False
```

Let's assume that "ToFrench" is a method that translates English to French. If you step through this code using the Step Over menu item, the second line of code is executed, but the Debugger won't display the code in the ToFrench method. It executes the ToFrench method and continues with the next line of code.

Step In

Step In executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger displays the method and steps through the method's code. When the method is finished executing, the Debugger returns to the calling method or event handler and continues with the next line of code.

Step Out

Step Out executes the rest of the method without stopping on each line. This is handy when you have used Step In to step through a method that was called by another method and now wish to continue code execution without stopping on each line. If you entered the current method or event handler using Step In, then stepping out executes the rest of the method and stops on the next line of code in the method that called the method you are stepping out of.

Tracking Method Execution with the Stack

A method or event handler can call another method or event handler which can call another one. This can go on for a while and you may need to keep track of the path of methods that were executed to get you where you are now. The Stack drop-down list does just that. When code execution begins (for example, when a PushButton is clicked), the Stack lists the PushButton's action event handler. If the action event handler calls a method, that method is added to the top of the list in the Stack when it's called. If that method calls another method, it is added to the top of the list. Once the current method finishes executing, it is removed from the list as REALbasic returns to the method that called it. It's called the "Stack" because the methods are "stacked" one on top of the other in the order they were called.

If you need to see the code from a method or event handler called earlier in the stack, click on its name in the Stack drop-down list to display the code for that method in the Code Editor area of the Debugger screen.

NOTE: The larger the Stack gets, the more memory is being used. If you run out of memory it could be because your stack is so long that it takes up all the memory that has been allocated to the stack. The solution is try to make fewer method calls and use fewer local variables.

Watching Your Values

Part of debugging is monitoring the conditions under which certain lines of code execute. Another part of debugging is monitoring the values of variables, objects, and properties as your code executes. The Variables pane is used for these purposes. This pane displays any local variables, parameters, the current object, and its super class. It also displays global properties from modules and the Application subclass if there is one.

Local Values

The Variables pane displays the local variables with their current values or, in the case of object, links to their object viewers. Click a hypertext link to display an Object Viewer pane that shows the current values for all the properties of the object.

Negative numbers appear in red in an Object Viewer. Items that appear in blue underline are hyperlinks to their Object Viewers. They update as you step through your code and the code affects the values of properties.

NOTE: The Object Viewer currently only supports viewing single dimension arrays.

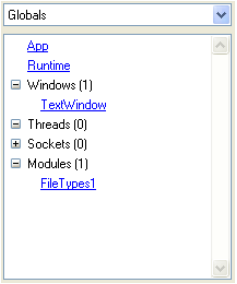
Parameters

If the Variables pane appears while REALbasic is executing a method that is passed parameters, the values of those parameters are shown in the Variables pane when the method is called.

Global Values

The top entry in the Variables pane is for global variables. It displays a viewer for items global to the application. This includes the App class, modules, threads, sockets, and windows. Any of these items that exist in the current application will have links to viewers for each item.

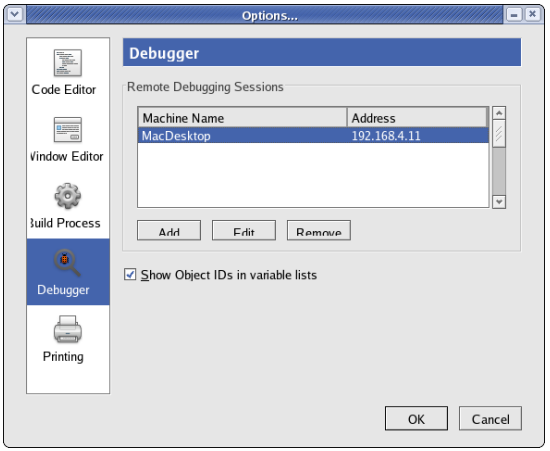
Figure 370. The Globals pane.



Object IDs

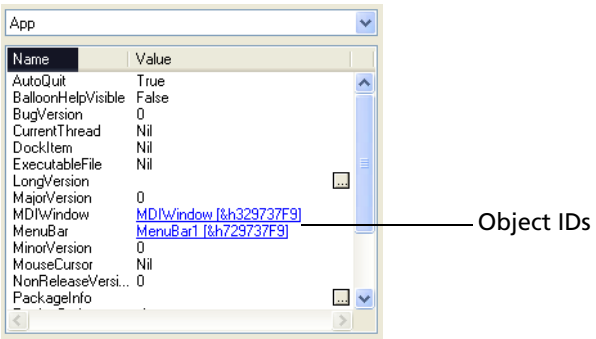
The Debugger can optionally display the internal object IDs within the Debugger. These IDs are used internally by the compiler and are not generally needed to debug REALbasic applications. For that reason IDs are not displayed by default. To display the object IDs, choose Edit ▶ Options (Linux and Windows) or REALbasic ▶ Preferences (on Mac OS X) and click on the Debugger icon in the browser area. The Debugger Preferences screen is shown in Figure 371.

Figure 371. Debugger Options.



Click the "Show Object IDs in Variable Lists" CheckBox and click OK to save your preference. When you use the Debugger, the object IDs will be shown in brackets following each object's data type or class, as shown in Figure 372. It shows the viewer for the App class, which will be in the Globals pane.

Figure 372. The Debugger with the Show Object ID preference selected.



Starting and Stopping Your Project

You can switch back to the Development environment while your application is running in the Debugger by clicking the Stop button in the Debugger toolbar or choose Project ▶ Stop (Ctrl+K or ⌘-K). Click the Resume button in the Debugger toolbar to resume execution.

Runtime Exception Errors

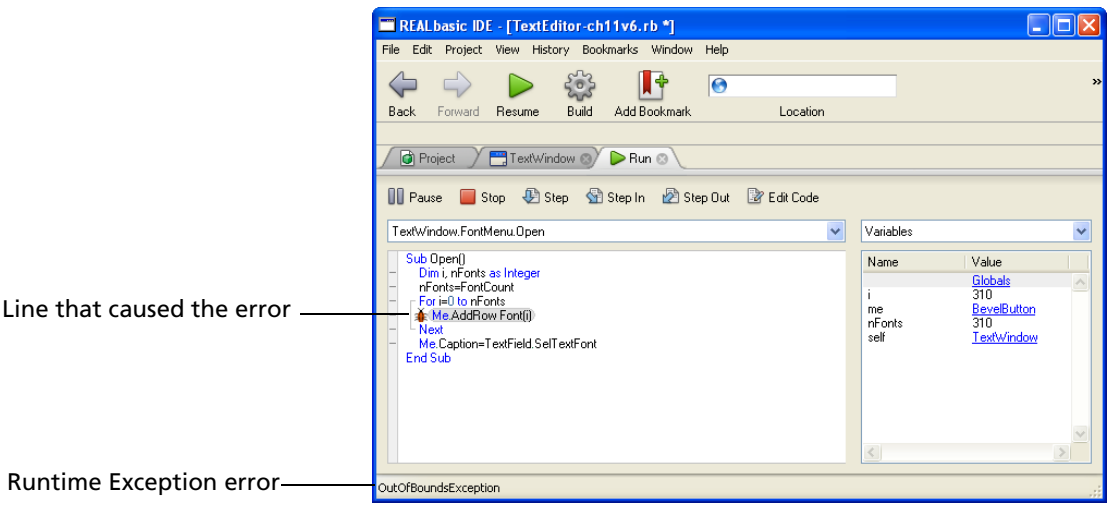
When you test your application in the IDE, REALbasic performs basic syntax checking and, if it finds a problem, it stops compilation and alerts you to the problem (see Figure 359 on page 500).

There are certain types of errors that can be detected only at runtime, (i.e., when the line of code that contains the problem is actually executed). This is because these errors depend on values that are not known until the code is running. An application containing a *runtime error* compiles without error and may even run without problems for a very long time before the lines of code containing the error are actually called.

But when these lines are executed, REALbasic has no choice but to stop execution. If you are testing the application in the IDE, the Code Editor will reappear and an error message will be shown below the line that contains the runtime error. An example runtime error message is shown in Figure 373.

To find and fix all such errors via the Debugger, you should select the Break on Exceptions item in the Project menu. With Break on Exceptions on, the Debugger will appear when a runtime error is encountered while you are testing your application. However, the unhandled exception will cause the built (standalone) application to quit.

Figure 373. An Unhandled Runtime Exception error in the Debugger.



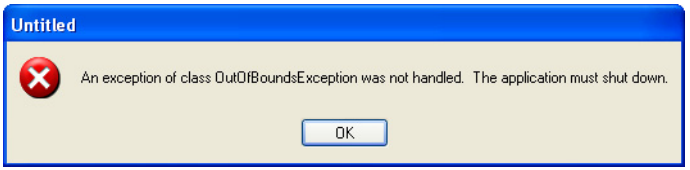
This error occurs when the value of the counter, i, reaches the value of nFonts. The programmer has forgotten that font numbering starts at zero rather than one; the loop should be from zero to FontCount-1. Until the value of nFonts is actually reached at runtime, the error cannot be detected. If the error is buried in an obscure method that is very rarely called, the application may survive many hours of testing without encountering the error.

As with syntax errors, the Tips bar contains the error message.

Runtime Errors in Standalone Applications

If the error occurs in a built application, REALbasic displays a generic message box that identifies the type of problem that it encountered. An example message box is shown in Figure 374.

Figure 374. An Unhandled Runtime Exception Error in a Built Application.



The application will quit when the end-user accepts the message box.

Handling Runtime Errors

REALbasic provides a way to detect and handle runtime errors that occur in standalone applications. There are several types of so-called *runtime exceptions* that you can detect in your code and handle. Please see the description of each Runtime Exception error in the *Language Reference* for more examples.

You can "catch" and handle runtime errors using the Try or Exception blocks. With an either type of block, you can display a more informative message box that will help track down the problem.

Exception blocks always appear at the end of a method (not where you think the error might occur) because every line after the Exception line is considered part of the exception block.

When a block 'catches' an exception error, the code in the Exception block runs, allowing you to obtain information on the location of the problem and the values that caused the error.

With an exception block, you can add code to your methods that handle the error in specific ways, depending on the type of error and the context in which it was encountered. This provides more control over the error handling process than the Break on Exceptions menu command.

In the Code Editor, the Exception line has the same level of indentation as the Sub or Function line. You can use **Exception** alone if you wish to handle any type of exception in the Exception block, as shown below:

```
Sub...
   .
   .
Exception
 MsgBox "Something really bad happened, but I don't know what."
```

The syntax of an Exception block is as follows:

```
Exception errorParameter As errorType
```

Both *errorParameter* and *errorType* are optional; *errorType* cannot be used without errorParameter.

The example shown above is sufficient to prevent the application from quitting, but the message is not very informative because you don't have a clue what type of exception occurred.

One way to test *ErrorParameter* is with an If statement in the Exception block:

```
Sub...
  .
  .
Exception err
  If err IsA TypeMismatchException then
    MsgBox "Tried to retype an object!"
  elseif err IsA NilObjectException then
    MsgBox "Tried to access a Nil object!"
  .
  .
  End if
```

For example, the runtime exception error shown in Figure 373 on page 511 can be handled with the following exception handler.

```
Sub Action()
  Dim i, nFonts as Integer
  nFonts=FontCount
    for i=1 to nFonts
      listBox1.addrow(Font(i))
    next
Exception err
  if err IsA OutOfBoundsException then
    msgBox "Error loading font names into Font menu!"
  end if
```

When this method runs, it displays a message box such as shown in Figure 375:

Figure 375. A Handled OutOfBoundsException error.

When the end user accepts this message box, the application does not quit.

Instead of using multiple If statements, you can also use multiple Exception blocks, each of which handles a different runtime exception type:

```
Sub...
  .
  .
Exception err as TypeMismatchException
  MsgBox "Tried to retype an object!"
Exception err as NilObjectException
  MsgBox "Tried to access a Nil object!"
```

In case you fail to include an Exception block in the method in which the error occurs (or a method that calls the defective method), you have one last chance to "catch" runtime errors before they bring down a built application. This is in the UnhandledException event of the Application class. This event occurs if a runtime error occurs anywhere in the application that was not handled by an Exception block. This event is passed a parameter of type RuntimeException and you can write a function that handles the exception. For safety's sake, you can write a generic routine that catches all runtime exceptions.

In this example, if the Action event shown here did not include an Exception block, you could catch the runtime error using the following function in an App class (a class derived from the Application class).

```
Function UnhandledException(error as RuntimeException) as Boolean
 If error IsA OutOfBoundsException then
  MsgBox "An OutOfBounds Exception error has occurred!"
 End if
 Return True
```

This method of handling runtime errors cannot provide the specific diagnostic feedback (such as Figure 375) because it will receive all OutOfBoundsExceptions throughout the application.

Remote Debugging

Remote debugging enables you to test your application on a different computer from your development machine. For example, if you are developing under Mac OS X and need to test the application under Windows, you can do so from your Macintosh. Remote Debugging sends a debug build of the application to the remote machine and launches it automatically.

Without Remote Debugging, you would have to create a standalone Windows build on your Macintosh, move it to the Windows computer, and then launch it from that computer.

Remote Debugging is available only in the Professional version of REALbasic.

We'll call the computer on which the REALbasic IDE is running the *development machine* and the remote computer on which you will be testing your application the *debugging machine*.

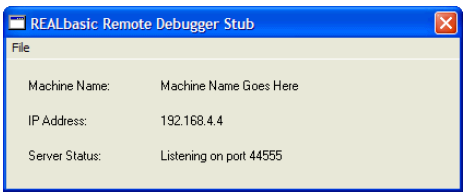
Remote Debugging works with a small utility application that runs on the debugging machine. It is called the Remote Debugger Stub. You first configure the Remote Debugger Stub on the debugging machine. When you want to test the application on that machine, you use the IDE's Project ▶ Run Remote menu command instead of the Project ▶ Run command.

To configure the Remote Debugger Stub, do this:

1 Copy the Remote Debugger Stub application to a debugging machine and double-click it.

The Stub main screen appears. Initially, the machine name will be "Machine Name Goes Here".

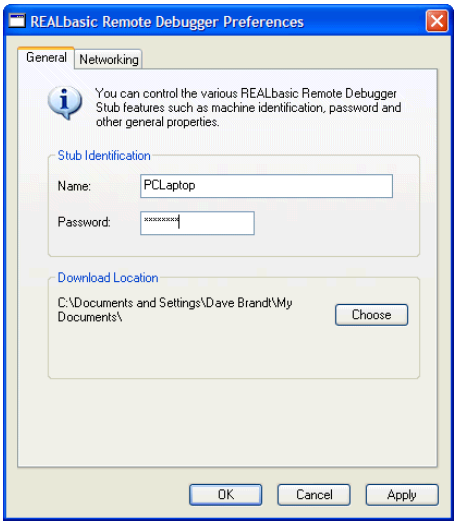
Figure 376. The Remote Debugger Stub main screen.



2 Choose Edit ▶ Options (Windows and Linux) or Remote Debugger Stub ▶ Preferences (Mac OS X) to configure the Remote Debugger Stub.

The configuration dialog box appears:

Figure 377. The Remote Debugger General Preferences screen.



It offers the following options:

- **Machine Name:** The name of the debugging machine that will identify it on the development machine. It can be any name you like.
- **Password:** The password you will use from the development machine to access this machine.

The second panel of the Preferences screen allows you to set the listening port. Normally, you do not need to change this preference.

- **Listening Port:** The port that all incoming TCP connections should be routed to. Most people will want to use the default value. If you have a firewall, you may wish to change this value.

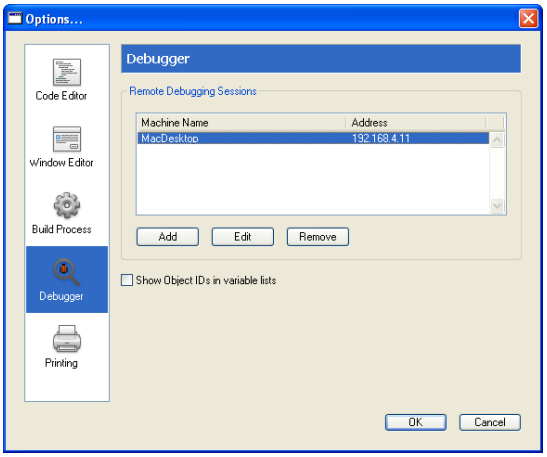
The next step is to set up Remote Debugging on your development machine. Before you can use a remote machine for debugging, REALbasic needs to be configured to send the debug application to the remote machine and launch it automatically.

To configure the REALbasic IDE for Remote Debugging, do this:

1 **Choose Project ▸ Run Remotely ▸ Setup.**

The Debugging panel of the Options dialog box appears. Alternately, you can open the Options dialog box directly (Preferences dialog on Macintosh) and navigate to Debugger options.

Figure 378. The Options Debugging panel.

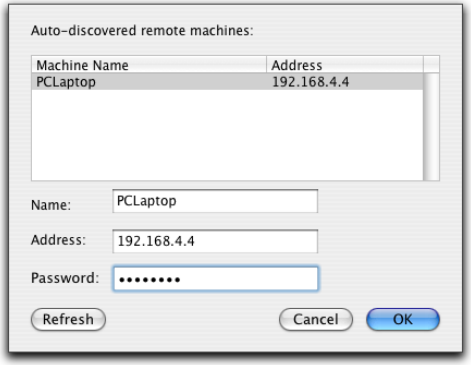


The Remote Debugging Sessions ListBox will show all the debugging machines that have been configured for remote debugging. You need to add the desired machines to this list.

2 **Click Add to add a debugging machine to the list.**

The Auto-discover dialog box appears.

Figure 379. The Auto-discover Remote Machines dialog.



In Figure 379, one computer is running the Remote Debugger Stub. It is identified by the Machine Name that had been entered into the Remote Debugger Stub configuration screen. If you find that not all your remote debugging machines are listed, verify that the Remote Debugger Stub is actually running on those machines and that you have selected the Public Debugger Stub option. This is set by default on the Networking preferences screen.

3 **Highlight a debugging machine.**

Its Machine Name and IP address appear in the Name and Address areas in the lower section of the dialog box. You can also add a debugging machine by typing its name and IP address into the entry areas but you may want to figure out why REALbasic was unable to see it automatically.

4 **Enter the password if the machine is password-protected.**

5 **Click OK to add the debugging machine.**

When you return to the Debugging Preferences panel, it will appear in the Remote Debugger Connections ListBox.

6 **If desired, repeat the process to add additional debugging machines.**

When you are finished with the Debugging Preferences screen, the selected debugging machines are added to the Project ▶ Run Remotely submenu. In this example, three Autodiscovered computers are shown:

If more than one debugging machine is available (as is this case here), the most recently used debugging machine will be at the top of the menu and get the keyboard shortcut.

To debug on a remote machine, do this:

1 **Verify that the Remote Debugger Stub is running on the desired debugging machine.**

2 **Choose the desired debugging machine from the Project ▶ Run Remotely ▶ *DebugMachineName* menu.**

REALbasic creates a debug build for the correct platform, sends it over the network to the debugging machine and launches it automatically. As it is doing so, progress indicator on the development machine gives you the status of the operation.

3 Reorient yourself in front of the debugging machine and test the application normally.

If you have put a breakpoint in your code or otherwise break into the Debugger, it will appear on the development machine.

When you are finished debugging, you need to quit out of the debug build before resuming work in the IDE on the development machine. You can do so on the debugging machine or, on the development machine, click the Stop button in the Debugger or choose Project ▶ Stop Debugging.

About Firewalls

In order for remote debugging to work, you will need to open two UDP ports in your firewall. You must allow both incoming and outgoing traffic to ports 44553 and 44554, as well as TCP port 13897 open on the IDE machine to debug if you have a firewall. You can pick the port that you want all TCP traffic to come in on, but these ports must be open to UDP traffic for the debugger to work.

CHAPTER 12

Communicating With The Outside World

Some applications need to communicate with other applications or even serial hardware devices to exchange information. Sometimes this is done automatically while other times it is initiated by the user. For example, when you use your computer to connect to the Internet, you are initiating communications between an application on your computer and an application on another computer at your Internet Service Provider (ISP). Fortunately, REALbasic provides controls that make communications between applications on different computers, and even communications between a computer and a serial hardware device easy.

Contents

Communicating With Serial Devices

A serial device is a device that communicates by sending and/or receiving data in serial. This means that it is either sending data or receiving data at any one moment. It doesn't send and receive at the same time. The most common serial device is a modem. Some printers are serial devices. Serial communications using REALbasic are done with the *Serial* control. To communicate with a serial device you configure a Serial control, open the serial port to make the connection, read and/or write data to and/or from the serial device connected to one of your serial ports, and finally close the serial port when you are through to disconnect from the serial device.

Getting Set Up

So the first step is to place a Serial control in one of your project's windows or instantiate a Serial object using code. Before you can begin communicating with a serial device using a Serial control, you need to set up the Serial control so that it will know which serial port your serial device is connected to. You will also need to set the speed at which communications will occur, as well as a few other settings. This can all be done at design time using the Properties pane or at runtime using code.

How you configure the Serial control's behavior properties will depend on what the serial device is expecting. Some devices can only communicate with one specific configuration. Other devices (like modems) can communicate using many different configurations. In the case of a modem, you will not only have to consider what configurations the modem will accept but also what configuration the modem your modem will be connecting to will accept. The Serial control's default configuration should work for most modems. You may need to change the default configuration for other serial devices.

Opening the Serial Port

Once you have configured the Serial control, you can open the serial port to initiate communications with the serial device. This is done by calling the Open method of the Serial control. This method is, in fact, a function that returns True if the connection is opened and False if it is not. For example, suppose you have a Serial control whose name is "Serial1." To open the serial port using this control, you can use the following code:

```
If Serial1.Open then
  MsgBox "The serial port was opened."
Else
  MsgBox "The serial port could not be opened."
End if
```

Once you have successfully opened the serial port, it will be unavailable to all other applications (and in fact, to other Serial controls as well) until it's closed.

Reading Data

When the serial device sends data back to the Serial control that is connected to it, the Serial control's DataAvailable event handler executes. The data that has been

sent back goes into a place in the computer's memory called a *buffer*. The buffer is simply a place to store the data that has been sent by the serial device because most serial devices don't have much memory of their own. When new data arrives in the buffer, REALbasic executes the DataAvailable event handler of the Serial control.

In the DataAvailable event handler, you use the Serial control's Read or ReadAll methods to get some or all of the data in the buffer. Both of these methods act as functions. Use the Read method when you want to get a specific number of bytes (characters) from the buffer. If you want to get all the data in the buffer, use the ReadAll method. In both cases, the data returned from the buffer is removed from the buffer to make room for more incoming data. If you need to examine the data in the buffer without removing it from the buffer, you can read the data from the Serial control's LookAhead method.

This example appends any incoming data to an EditField:

```
Sub DataAvailable()
  EditField1.Text=EditField1.Text+Me.ReadAll()
```

You can clear all data from the buffer without reading it by calling the Serial control's Flush method.

Both the Read and ReadAll methods of the Serial class take an optional parameter that enables you to specify the encoding. Use the Encodings object to get the desired encoding and pass it as a parameter. For example, the code above has been modified to specify that the incoming text uses the ANSI encoding, a standard on Windows:

```
Sub DataAvailable()
  EditField1.Text=EditField1.Text+Me.ReadAll(Encodings.WindowsANSI)
```

You may need to specify the encoding when text is coming from another platform or is in another language. For information about text encoding, see the section "Working with Text Encodings" on page 333.

Writing Data

You can send data to the serial device at any time as long as you have opened the serial port with the Serial control's Open method. You send data using the Serial control's Write method. The data you wish to send must be a string, as the Write method accepts only a string as a parameter.

The Write method is performed asynchronously. This means that the next line of code following the Write method can already be executing before all the data has actually been sent to the serial device. If you need your code to wait for all data to be sent to the serial device before continuing, call the Serial control's XmitWait method immediately following a call to the Write method.

Changing a Serial Control's Configuration on the Fly

There may be times when you need to change a Serial control's behavior properties while the serial port is open. While you can change these properties, the changes won't take effect until you close the serial port and reopen it. If you need the behavior properties to update immediately, call the Serial control's Poll method. This updates all properties immediately and calls the DataAvailable event handler immediately if there is any data waiting in the buffer.

Closing the Port

Once you are finished communicating with a serial device, you must close the serial port to end the communications session and make the port available to other Serial controls or other applications. To close the serial port, call the Close method of the Serial control that opened the serial port.

Communicating With Modems

Modems have a set of commands you can send them to tell the modem to do things such as dial a particular number. Most of these commands are the same for every modem. Your modem probably came with a guide that lists its commands. Consult that guide for more information.

Communicating with USB and FireWire Devices

These devices use a much higher-level API that requires drivers for interaction. If you'd like to control such a device, you will need to obtain a shared library from the manufacturer or write your own.

TCP/IP Communications with the TCPSocket Control

Sometimes applications need to communicate with other applications on the same network. This can be accomplished using the REALbasic's *TCPSocket* control. The TCPSocket control can send and receive data using TCP/IP.

In versions of REALbasic prior to 5.0, TCP/IP communications were handled with the Socket control. The Socket control has been superseded by the TCPSocket control. The icon in the Controls list that previously added a Socket control to a window or project now adds a TCPSocket control.

The TCPSocket control is derived from a new base class, the SocketCore class. The SocketCore class handles the core functionality that both the TCP and UDP protocols provide (the UDP protocol is described in the section "UDP Connections with the UDPSocket Control" on page 531). The SocketCore class is an abstract class that cannot be instantiated by itself. The SocketCore class has properties and methods that are inherited by the TCPSocket, ServerSocket, and UDPSocket controls. If you have used the Socket control in your projects, you should update to the TCPSocket control.

TCP/IP is the protocol of the Internet. It's the way most data is transmitted via the Internet. In fact, the "IP" in TCP/IP stands for "Internet Protocol."

The TCPSocket control can be used to communicate with other computers on the same network. When you connect to the Internet, you are part of the Internet

network. This allows you to communicate with other computers on the Internet via TCP/IP.

Getting Set Up

You can either add a TCPSocket control to a window or instantiate a TCPSocket object using code. Before you can connect to another computer using the TCPSocket control, you must first set the *port*. The port is to TCP/IP what channels are to television or frequency assignments are to radio stations. Ports give an application the ability to focus on specific data rather than receiving all the data transmitted to your computer via TCP/IP. This allows you to browse the web and send email at the same time because the web uses one port and email uses another. The port is represented by a number and there are thousands of available ports. Some have already been designated for specific functions like web browsing, email, FTP, etc. If you are designing an application that will need to communicate with another application, you will need to find out what port the other application is using. For example, if the other application is an SMTP server, it's probably using port 25 since that is the port that is reserved for SMTP (Simple Mail Transfer Protocol).

A TCPSocket control has a Port property (inherited from the SocketCore class) that can be assigned at design time or runtime but it must be assigned a value before you can connect to another computer. If you plan on initiating the connection, you must also assign the IP address of the computer you wish to connect to the Address property of the TCPSocket control that will make the connection.

Mac OS X and Linux have a built-in restriction regarding port numbers. Ports below 1024 cannot be assigned by a user who is not running with "root" privileges. Mac OS X is configured so that a user cannot gain root privileges via the graphic user interface. Most users run with Admin privileges—not root—so you should use ports above 1024 for normal TCP/IP communications with Mac OS X computers because the TCPSocket cannot access port numbers below 1024. This is not a bug but a security feature that is built into these operating systems.

NOTE: A TCPSocket control can only be connected to one application at a time. If you need to maintain multiple connections simultaneously, you should use the ServerSocket control, which is designed for this purpose. See the section "Handling Multiple Connections with the ServerSocket Control" on page 529.

Making a Connection to Another Computer

Once you have assigned a port and an IP address, you can connect to an application on the computer at that IP address, provided that the application is listening for TCP/IP connections on the port you have specified. To initiate a connection, you call the TCPSocket control's Connect method. A connection is not necessarily established immediately after you call the Connect method. There may be network connection issues that take some time to resolve before the connection is actually established.

There are two ways to determine that you are connected. You can either wait until you receive a Connected event from your TCPSocket or test the return value of the IsConnected method of the SocketCore class. If you don't wait for this feedback, you

either cause the connection process to halt, resulting in a lost connection error (102), or an out of state error (106).

When the connection is established, the TCPSocket control's Connected event handler executes. If a connection is not established, an error occurs and the TCPSocket's Error event handler is executed.

Once a connection is established, your application can begin sending and receiving data with the application at the other end of the connection.

Listening For a Connection From Another Computer

In some cases you may want your application to wait for another application to connect to it rather than initiate the connection. To do this, you use the TCPSocket control's Listen method. For example you have a button that when pressed, causes the application to listen for a TCP/IP connection on the port number that is assigned to the TCPSocket's Port property. Let's assume that the TCPSocket control is named "TCPSocket1." In the PushButton's Action event handler, you use the following code:

```
Sub Action()
  TCPSocket1.Listen
```

Once a connection is established, the TCPSocket control's Connected event handler executes, letting you know that you have a connection.

Reading Data

When the application at the other end of the connection sends data back to the TCPSocket control that it's connected to, the TCPSocket control's DataAvailable event handler executes. The data that has been sent back goes into a place in the computer's memory called a *buffer*. The buffer is simply a place to store the data that has been sent by the other application. When new data arrives in the buffer, REALbasic executes the DataAvailable event handler of the TCPSocket control.

In the DataAvailable event handler, you can use the TCPSocket control's Read or ReadAll methods to get some or all of the data in the buffer. Both of these methods act as functions. Use the Read method when you want to get a specific number of bytes (characters) from the buffer. If you want to get all the data in the buffer, use the ReadAll method. In both cases, the data returned from the buffer is removed from the buffer to make room for more incoming data. If you need to examine the data in the buffer without removing it from the buffer, you can read the data from the TCPSocket control's LookAhead property.

This example appends any incoming data to an EditField:

```
Sub DataAvailable()
  EditField1.Text=EditField1.Text+Me.ReadAll()
```

When you are reading text from an outside source, you may need to specify the text encoding. The text encoding is the scheme that maps each letter in the text to a numeric code. If the text comes from another application, operating system, or is in

another language, you may need to tell REALbasic which encoding was used. For more information on text encoding, see the section "Working with Text Encodings" on page 333.

Both the Read and ReadAll methods of the TCPSocket class take an optional parameter that enables you to specify the encoding. Use the Encodings object to get the desired encoding and pass it as a parameter. For example, the code above has been modified to specify that the incoming text uses the ANSI encoding, a standard on Windows:

```
Sub DataAvailable()
  EditField1.Text=EditField1.Text+Me.ReadAll(Encodings.WindowsANSI)
```

From then on, REALbasic stores the encoding with the text. The text will display and print properly and string operations will work as expected.

Writing Data

You can send data to the application you are connected to at any time. You send data using the TCPSocket control's Write method. The data you wish to send must be a string, as the Write method accepts only a string as a parameter. In this example, the text from an EditField is being sent via a TCPSocket control:

```
TCPSocket1.Write EditField1.Text
```

If you need to send the text to an application that is expecting a specific text encoding, then you should convert the text to that encoding prior to sending it. Use the ConvertEncoding function to do this. Its parameters are the text to be converted and the text encoding to use. For example, the following line writes the text in the EditField using the MacRoman encoding:

```
TCPSocket1.Write ConvertEncoding(EditField1.text,Encodings.MacRoman)
```

When you call the Write method, you begin the process of sending data across the network. Certain low-level socket service providers have limits on the maximum amount of data the socket can send in one batch. This is dependent on a few factors; among them are which library is providing the TCP/IP services (such as Open Transport on Macintosh Classic, WinSock on Windows, etc.), and how much data you are trying to send.

You might think that the provider will only affect transfers of large data, but this is not true. Never assume how much data REALbasic will send between calls to the SendProgress event. It is normal to see this fluctuate. Each provider specifies the minimum and maximum amount of data it will send.

If you are trying to send data larger than the maximum, it will not be sent all in one chunk. Instead, REALbasic will loop until your data is completely sent, giving you periodic SendProgress events. If you try to send too little data, the internet service provider (ISP) will queue your data up. This doesn't always mean your data has been sent although you will receive SendProgress and SendComplete events.

This is due to the ISP implementing the Nagle algorithm, which helps network productivity. For every chunk of data that is sent across the network, there is a header attached to the beginning of that data. You never will have to deal with these headers, because they are taken care of for you by the ISP. The reason this is important, though, is that if you are sending one and two bytes at a time across the network, you are also attaching these 40 or so bytes of header to each send. This can bog down a network unless the Nagle algorithm is implemented.

Currently, REALbasic does not allow the user to turn this feature off, and leaves it set to the default. Note that, if you send only one byte of data, and never send anymore, the system will still send your one byte out even though the Nagle algorithm is enabled.

The conclusion is this: Do not assume that you know when your data has been completely sent. Rely on the SendComplete event to tell you when the send has finished. Also, do not expect the bytesSent parameter of the SendProgress event to be the same value every time. This value will change based on how many bytes of data the ISP was able to send.

Note: If you are going to be sending small chunks of data across a network (especially a small network), it might make more sense to use the UDPSocket class instead.

Handling Errors

Errors can occur attempting to connect, sending data, or receiving data. Errors are not always what they seem. For example, when the other computer closes the connection, an error is generated. When an error occurs, the TCPSocket control's Error event handler is executed. Errors are represented by numbers. The TCPSocket control's LastErrorCode property will contain the number of the last error that occurred. See the SocketCore class in the *Language Reference* for a complete list of error numbers.

Errors are simply ways to alert your application to conditions it may not have anticipated or be able to anticipate. For example, if you attempt to make a connection in Mac OS 8-9 or listen for one and you don't have Open Transport installed, an error is generated.

Orphaning a Socket

One of the new features of the new socket architecture is the ability to orphan a socket. This feature is needed to support the functionality of the ServerSocket.

To make this new functionality possible, we have introduced some new behavior to the socket class. When you call the Connect method of the SocketCore class, the Listen method of the TCPSocket class, or add a socket using the AddSocket method of the ServerSocket class, your socket's reference count is incremented.

This means that the socket does not have to be owned by the window in order for it to continue functioning. This is helpful in certain circumstances. For example, suppose you write your own socket subclass that implements all of the events for the

socket, called MySpiffySocket. In the action event for a PushButton you use the following code:

```
Dim s as MySpiffySocket

s = New MySpiffySocket
s.port = 7000
s.address = "somecool.server.com"
s.connect
```

The socket will continue to stay connected, even though there is nothing owning a reference to it except within the PushButton's Action event.

In other words, a socket will continue to live until you tell it to die. If you have dragged a TCPSocket to a window and then called the Connect or Listen methods before closing the window, there will be two references to the socket, whereas in previous versions of REALbasic there was only one reference—the window's reference. In this case, the socket will continue to function until its connection is terminated, even after the window has been closed.

The termination can be done either locally, by calling the Close method of TCPSocket or remotely, with the remote host terminating the connection. If you have not called the Connect or Listen methods, then there will be only one reference to it (the window's), and it will be destroyed appropriately when the window closes. In either case, once the application terminates, all sockets are released gracefully, and your application will not leak memory.

Maximum number of Sockets

There is a limit to the number of sockets your application can have opened concurrently on Mac OS X (prior to Mac OS X 10.3). This is because BSD sockets use a file descriptor for each open socket (one that is currently bound to any port on the machine). The standard limit on Mac OS X is set to 256 file descriptors, but this limit can fluctuate based on the amount of RAM in your machine. This means that you can have, at most, 256 sockets connected at once per application. In practice, this number tends to be less than 256, because your application might have files open, or the underlying API calls might be using a file descriptor for their purposes. This is not an issue on Windows, Mac OS 9 or (to a certain extent) Linux, and it is not a bug in REALbasic. It is a characteristic of the underlying BSD system.

Note that you can run into this issue on Linux, but it tends to be far less likely.

Closing the Connection

When you are finished communicating and wish to disconnect from the other application, you do so by closing the connection. The connect is closed by calling the TCPSocket control's Close method. Suppose you have a Socket named "Socket1" that has established a connection. To close the connection, you can use the following code:

```
Socket1.Close
```

Sending and Receiving Email via TCP/IP

REALbasic includes subclasses of the TCPSocket class that are designed specifically to build an email client application—an application that sends and receives email, like Apple's Mail program, Eudora, or Microsoft's Entourage.

A typical email client program actually uses two protocols, POP, which stands for Post Office Protocol and SMTP, which stands for Standard Mail Transfer Protocol. The POP protocol is used to receive messages and SMTP is used to send messages. An email client program uses both to communicate with a mail server that supports POP and SMTP, such as 4D Mail.

REALbasic provides direct support for these two protocols via the POP3Socket and the SMTPSocket classes. Both are subclassed from TCPSocket but have special methods and properties to make it easy to handle email.

For example, the POP3Socket class has properties for the Username and Password sent to the mail server and methods to poll the mail server for unread messages, retrieve messages, delete messages, and disconnect from the mail server.

The SMTPSocket class has a property to hold the messages in queue to be sent and methods to add email messages to the queue, send messages, and disconnect from the mail server.

To manage an email message, REALbasic uses three additional classes, EmailMessage, EmailHeaders, and EmailAttachment. The EmailMessage class holds the body, header, and attachments, which are created or stored using the EmailHeaders and EmailAttachment classes.

For detailed information on the methods and properties of these classes, see the Language Reference. For an example email client, see the example email application on the REALbasic CD or the REALsoftware web site.

The Professional version of REALbasic supports secure POP3 and SMTP communications via the POP3SecureSocket and SMTPSecureSocket classes. These classes are subclassed from the SSLSocket class, but are otherwise identical. You can support secure communication by setting the Secure property of the SSLSocket class to True.

HTTP Communications

The HTTP protocol is the protocol that web browsers use. REALbasic supports the HTTP protocol via the HTTPSocket class, derived from the TCPSocket class, and the HTTPSecureSocket class, derived from the SSLSocket class. It contains methods and properties that enable you to retrieve a URL or post a form using HTTP.

With the Professional version of REALbasic, you can support secure communications (https) by invoking the Secure property of the SSLSocket class. When you do so, the default port changes from 80 to 443.

See the *Language Reference* for information about the methods, events, and properties of these two classes.

Handling Multiple Connections with the ServerSocket Control

If you need to communicate with more than one application via the same port, it is difficult to do using the TCPSocket because each TCPSocket can manage only one connection at a time. To use the TCPSocket, you would have to implement a system for managing multiple TCPSockets, as connections are received.

To handle this situation, you should use the ServerSocket control. A ServerSocket is a permanent socket that listens on a single port for multiple connections. When a connection attempt is made on that port, the ServerSocket hands the connection off to another socket, and continues listening on the same port. Without the ServerSocket, it is difficult to implement this functionality due to the latency between a connection coming in, being handed off, creating a new listening socket, and restarting the listening process. If you had two connections coming in at about the same time, one of the connections may be dropped because there was no listening socket available on that port.

To initiate the ServerSocket's listening process, set the port to listen to by assigning a value to the Port property and call the ServerSocket's Listen method.

On Mac OS X and Linux, attempting to bind to a port less than 1024 will cause a SocketCore.Error event to fire with an error 105 unless your application is running with root permissions. This is a built-in security feature of Unix-based operating systems. This is not a bug, but a security feature that prevents problems that can arise from allowing sockets to listen on privileged ports.

The ServerSocket control automatically manages a "pool" of TCPSockets available for use. You don't create the TCPSockets explicitly; instead you can set the size of this pool using the MinimumSocketsAvailable and MaximumSocketsConnected properties after establishing the listening socket. If you change the MaximumSocketsConnected property, it will not kill any existing connections (it just may not allow more connections until the existing connections have been released). If you change the MinimumSocketsAvailable property, it may fire the AddSocket event of the ServerSocket to replenish its internal buffer.

When you call the ServerSocket's Listen method, it first fills its internal pool of handoff sockets. It does this by calling the AddSocket event. This event will be called until it has enough sockets in the internal pool of available sockets. It adds the number specified by the MinimumSocketsAvailable property, plus ten extra sockets. (Note that if you return Nil from this event, it will cause a NilObjectException.) The ServerSocket is not ready to hand off connections until this process is completed. Connections that come in while the server is populating its pool are rejected. To determine when the ServerSocket is ready to accept incoming connections, check its IsListening property.

A ServerSocket can only return a TCPSocket (or a subclass of TCPSocket) in its AddSocket event. Since UDP is a connectionless protocol (see "UDP Connections

with the UDPSocket Control" on page 531), it does not make sense for a ServerSocket to deal with UDPSockets.

Reference Counting

The reference count isn't incremented when you return a socket with the AddSocket method. Instead, the socket is pooled internally, and its reference count is incremented when the server hands off a connection to that socket. If the ServerSocket is destroyed before it uses one of these pooled sockets, the unused sockets get destroyed as well. Until that time, the ServerSocket is the parent of the TCPSocket, and so the TCPSocket will remain. If a socket returned from the AddSocket event has been handed a connection, and then the ServerSocket is destroyed, the socket will remain connected and continue to function.

The new functionality described here tells you that a socket can be orphaned. This does not hold true for a ServerSocket or a UDPSocket. Each of these sockets must have a reference holder. If it does not, then once the socket goes out of scope in your code, it is destroyed. However, when the ServerSocket is destroyed, it will not terminate any of your already-made connections (if there are any). It will only destroy TCPSockets that have not been connected.

The ServerSocket is a Professional-only feature of REALbasic.

Handling Secure TCP Connections with the SSLSocket Control

The SSLSocket control is used to do secure communications via TCP/IP using Secure Sockets Layer (SSL) technology. SSL is an internet protocol that specifies how to pass secure communications over the internet. This provides world-class security, allowing you to accept credit card numbers, serve medical records, human resources information, or other sensitive data without fear of interception.

There are currently four different protocols supported by REALbasic. They are shown below:

SSL v2	SSSL version 2
SSLv23	SSL version 3, but can roll back to version 2 if needed
SSLv3	SSL version 3
TLSv11	TLS (Transport Layer Security version 1

The default protocol is SSLv23; this is compatible with most SSL servers. You must set this property before you establish the connection by calling the Connect method of the SSLSocket class. Trying to set the protocol after you have begun the connection process will have no effect.

Not all servers will accept a connection with the default protocol (SSLv23). This is server-specific, and may not be known beforehand. When you don't know whether the server will accept SSLv23 connections, try making multiple connection attempts to the server. If the initial attempt is rejected and returns a 102 error, then try again with a different ConnectionType. Be sure to have a way to terminate this process if none of the connection types works, or if you get an error other than 102.

To establish an SSL connection, set the Secure property to True and use the Connect method.

Currently, setting the Secure property to True and then calling the Listen method will not create a secure listening socket. Doing so will create a non-secure listening port on your machine at the port specified, and any connections received will not be secure. A secure listening feature may be added later to REALbasic.

Like the ServerSocket control, the SSLSocket control does not have an icon of its own in the Controls list. Since it is not derived from the Control class, you can instantiate it via code. Or, you can add it to a window using the window's contextual menu by choosing Add ▶ SocketCore ▶ TCPSocket ▶ SSLSocket or drag a TCPSocket to a window and then change its Super Class to SSLSocket.

The SSLSocket control is a subclass of TCPSocket and is a Professional-only feature.

UDP Connections with the UDPSocket Control

The User Datagram Protocol, or UDP, is the basis for most high speed, highly distributed network traffic. It is a connectionless protocol that has very low overhead, but is not as secure as TCP. Since there is no connection, you do not need to take nearly as many steps to prepare when you wish to use a UDP socket.

Like the new TCPSocket control, the UDPSocket control is derived from the new SocketCore class. Unlike TCPSocket, the UDPSocket does not have an icon of its own in the Controls list. But, since it is derived from the SocketCore class rather than the Control class, you don't need to add a UDPSocket control to a window to instantiate it. You can instantiate it with code via the New operator.

In order to use a UDPSocket, it must be bound to a specific port on your machine. Once the bind has occurred, the UDPSocket is ready for use. It will immediately begin accepting any data that it sees on the port it has bound to. It also allows you to send data out, as well as set UDP socket options.

Datagrams

In order to differentiate between which machine is sending you what data, a UDPSocket uses a data structure known as a *Datagram*. A Datagram has two properties, *Address*, which is the IP address of the remote machine that sent you the data, and *Data* — the actual data itself. When you attempt to send data out, you must specify information in the form of a Datagram. This information is the remote address of the machine you want to receive your packet, the port it should be sent to, and the data you wish to send the remote machine.

UDPSocket Modes

UDP sockets can operate in various modes, which are all very similar, but have vastly different uses. The mode that most resembles a TCP communication is called "unicasting." This occurs when the IP address you specify when you write data out is that of a single machine. An example would be sending data to

www.realsoftware.com, or some other network address. It is a Datagram that has one intended receiver.

The second mode of operation is called "broadcasting." As the name implies, this is a broadcasted write. It is akin to yelling into a megaphone. Everyone gets the message, whether they want to or not. If the machine happens to be listening on the port you specified, then it will receive the data on that port. This type of send is very network intensive. As you can imagine, broadcasting can create a huge amount of network traffic. The good news is, when you broadcast data out, it does not leave your subnet. Basically, a broadcast send will not leave your network to travel out into the world. When you want to broadcast data, instead of sending the data to an IP address of a remote machine, you specify the broadcast address for your machine. This address changes from machine to machine, so REALbasic provides a property of the UDPSocket class which tells you the correct broadcast address.

The third mode of operation for UDP sockets is "multicasting." It is a combination of unicasting and broadcasting that is very powerful and practical. Multicasting is a lot like a chat room: you enter the chatroom, and are able to hold conversations with everyone else in the chatroom. When you want to enter the chatroom, you call JoinMulticastGroup, and you only need to specify the group you wish to join. The group parameter is a special kind of IP address, called a *Class D IP*. They range from 224.0.0.0 to 239.255.255.255. Think of the IP address as the name of the chatroom. If you want to start chatting with two other people, all three of you need to call JoinMulticastGroup with the same Class D IP address specified as the group. When you wish to leave the chatroom, you only need to call LeaveMulticastGroup, and again, specify the group you wish to leave. You can join as many multicast groups as you like, you are not limited to just one at a time. When you wish to send data to the multicast group, you only need to specify the multicast group's IP address. All people who have joined the same group as you will receive the message.

Multicasting has some extra features that make it an even more powerful utility for network applications. If you wish to receive the data you sent out with a multicast send, you can set the SendToSelf property (also known as loopback) of the UDPSocket. If it is set to True, then when you do a send (to a multicast group) you will get that data back. You can also set the number of router hops a multicast datagram will take (also known as the Time to Live). When your Datagram gets sent out, it runs through a series of routers on the way to its destinations. Every time the Datagram hits a router, its RouterHops property gets decremented. When that number reaches zero, the Datagram is destroyed. This means you can control who gets your datagrams with a lot more precision. There are some "best guesses" as to what the value of RouterHops should be.

Value	Description
0	Same host

Value	Description
1	Same subnet
32	Same site
64	Same region
128	Same continent
255	Unrestricted

Note that if your Datagram runs through a router that does not support multicasting, it is killed immediately.

Due to the connectionless functionality of UDP, it does not make any guarantees that your data will reach its destination. You can work around this by creating your own protocol on top of the UDP protocol which acknowledges receives.

The UDPSocket operates in asynchronous mode, but the Connect method works synchronously. If the connect fails, you will get an error event immediately, and the IsConnected property will be set immediately.

Making Networking Easy

If you only need to establish network communications among REALbasic applications on a network, you can use a group of classes that are specialized for this task. They are based on the TCP or UDP protocols but they can only communicate among REALbasic applications. If you need to communicate with another application, such as an FTP server, you need to use the more generic classes.

In exchange for this limitation, it is extremely easy to set up communications. Only a few lines of code are needed, and you don't need to know a lot about the TCP or UDP protocols.

The AutoDiscovery Class

The AutoDiscovery class is perfect for managing network communication among REALbasic applications. As its name implies, the AutoDiscovery class can automatically 'discover' all of the computers on the network that are running the 'easy' UTP protocol that's used as the basis for network communications with this class. With the AutoDiscovery class, you can send and receive messages with individual users or multicast to a group of users.

Joining a MultiCast Group

To join a multicasting group of users, all you need to do is pass the name of the group to the Register method of the AutoDiscovery class. Everyone who joins the group uses the same name. Internally, the class takes care of assigning a proper multicasting IP address for you. For example, the line:

```
AutoDiscovery1.Register("myMulticastingGroup")
```

is all you need to join the group. To get the list of all group members, call the GetMemberList method. It returns a string array of the IP addresses of all

computers currently in the group. At any time you can call the UpdateMemberList method to refresh this list.

This code gets the list of member computers and displays them in a ListBox.

```
Dim s(-1) as string //declare the array and let GetMemberList resize it
Dim i as Integer
s= AutoDiscovery1.GetMemberList
For i=0 to Ubound(s)  //Ubound gets index of the last element of s
  ListBox1.AddRow s(i)
Next
```

Sending a Message

To send a message to the group, you use the SendMessageToGroup method. It takes an integer command code (which you can assign arbitrarily) and the text of the message. For example, the following sends the contents of EditField2 as the message with a command ID of 50:

```
AutoDiscovery1.SendMessageToGroup(50,EditField2.Text)
```

To send a message to an individual, you need the computer's IP address, which you can get from the GetMemberList method, the command ID, and, of course, the message text. Call the SendMessageToIndividual method. For example, the following sends the same message to the person at 192.168.1.118:

```
AutoDiscovery1.SendMessageToIndividual("192.168.1.118",_
            ,50,EditField2.Text)
```

This example uses the underscore character to continue the line of code in a second line in the Code Editor.

The EasyUDPSocket and EasyTCPSocket Classes

The EasyUDPSocket and EasyTCPSocket classes are 'easy' versions of the UDPSocket and TCPSocket classes, respectively, that are specialized for REALbasic-to-REALbasic network communication. They both eliminate the need to deal with some lower-level networking issues in order to provide a simple solution to a particular problem.

The main difference between the EasyTCPSocket class and the regular TCPSocket class is the message-based aspects of the protocol. The connection process is identical to a regular TCPSocket. However, when you want to send data to a remote machine you must use the SendMessage method to do so. When you receive a message sent via this protocol, you will get a ReceivedMessage event.

Receiving a Message

Receiving a message couldn't be easier. The AutoDiscovery class has a ReceivedMessage event that fires when AutoDiscovery receives a message from anyone in the group (or an individual), including yourself. The event returns the IP address of the

sender's computer, the Command ID, and the message text. An event handler like this is all you need:

```
Sub ReceivedMessage(FromIP as String, Command As Integer, data as String)
 MsgBox FromIP + " sent us " + Str(Command) + ": " + data
```

Understanding Protocols

Any kind of communication requires that all parties involved agree on a method of communication and a language. For example, if you want to communicate with a friend, you might go talk to them face to face, call them on the phone, or send them email. Both of you must be able to communicate using the same language or you won't be able to communicate at all. Communications via TCP/IP work the same way. The language used is called a *protocol*. A protocol is simply an organized way of sending and/or receiving information.

If you are writing an application that will communicate with another application via TCP/IP, you will need to understand the protocol the other application will be expecting in order to communicate with it. For example, on the Internet, the protocol for the world wide web is called HTTP (HyperText Transfer Protocol), the protocol for sending email is called SMTP (Simple Mail Transfer Protocol), and the protocol for receiving email mail is called POP3 (Post Office Protocol 3). Complete descriptions of these Internet protocols and others are available on the Internet. The descriptions of these protocols are called RFCs (Request For Comments). The easiest way to find information on RFCs is to go to www.yahoo.com and search for "RFC". This will give you a list of links to various web sites that explain all of the various Internet protocols.

If you are writing an application that communicates with other applications you have written, then you can define your own protocol. Your protocol will simply be a set of commands you define that allow the applications to understand what the other wants.

CHAPTER 13

Extending the Capabilities of REALbasic

One of the things that makes REALbasic easy to learn and use is that it abstracts you from the inner workings of the operating system. You don't have to know any of the thousands of commands that make up the API (application programming interface) used to work with your computer's OS. This also means that REALbasic may not have a particular capability that you require. Fortunately, REALbasic provides several ways to extend its capabilities, allowing you to add just about any functionality you need.

Contents

Making API calls to the Operating System

Using the Declare statement, you can access the API on either the Macintosh or Windows platforms. PPC and Intel machines are supported. You need to use the conditional compilation feature to isolate your Declare statements for each platform. With the Declare statement, you specify the name of the toolbox call and its shared library, and the parameters the call uses. If the call returns a value, you specify the data type of the value that is returned.

If the functionality is available on both platforms, you can use the same name for both platforms. However, often the parameters for the call will be different. Use conditional compilation to isolate your calls as well.

The following button Action uses the Macintosh Speech manager to speak the text in an EditField:

```
dim s as string
dim i as integer
#if TargetMacOS then
 Declare Function SpeakString lib "SpeechLib" (SpeakString as pstring) as Integer
#endif
s=editField1.text
#if TargetMacOS then
 i=SpeakString(s)
#else
 MsgBox "Speech is supported only on Macintosh!"
#endif
```

If the name of the toolbox call is the same as a REALbasic method, use the Alias keyword to refer to the call. For example, if SpeakString was the name of a REALbasic method, you could not use the above syntax. You could use, for example:

```
Declare Function MySpeakString lib "SpeechLib" Alias "SpeakString"
(SpeakString as pstring) as Integer
```

You would then use MySpeakString in your code to invoke the toolbox call.

See the description of the Declare statement in the *Language Reference* for more information and examples.

Calling AppleScripts

AppleScript is Apple Computer's system-level scripting language that makes controlling applications easy. REALbasic supports AppleScript. You can write a script in AppleScript and then call that script in your REALbasic project.

Note: Applescript is available for use only on Macintosh.

Preparing an AppleScript to Work in REALbasic

In order for REALbasic to run an AppleScript, the entire script must be enclosed in an "on run" handler like this:

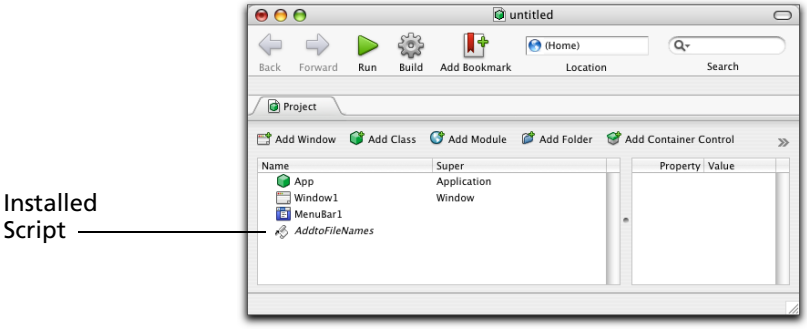
```
on run
  //your script code goes here
end run
```

Next, your script must be saved as a compiled script. In the Script Editor supplied by Apple Computer, choose File ▶ Save As. Then choose Compiled Script from the Kind popup menu.

Adding an AppleScript to a Project

To include an AppleScript, just drag your compiled script file into the REALbasic Project Editor. The script will appear with a script icon next to it. Figure 380 shows an example of a project with a script installed.

Figure 380. A compiled AppleScript in Project Editor



When you drag a compiled script into your Project Editor, REALbasic creates an alias to the AppleScript on disk. Its name is shown in italics, as indicated in Figure 380. When you create a standalone application, the Applescript is included in the built application.

Passing Values To an AppleScript

If you are writing a script you want to pass parameters to, the parameters must be enclosed in curly braces following the on run statement. In the following example,

the x and y are parameter variables that will hold the values of the two parameters passed to the script:

```
on run {x,y}
  //your script code goes here
end run
```

Returning Values From an AppleScript

You write a script to act as a function by having it return a value. To return a value from a script, simply use the return command in AppleScript followed by the value you wish to return. This simple example takes a number of days and returns the equivalent number of years:

```
on run {daysOld}
  return daysOld/365
end run
```

Calling an AppleScript

Scripts are called just like the built-in methods and functions. Type the name of the script as it appears in the Project Editor. If the script requires parameters, the parameters follow the name of the script just as they do with any of the built-in commands. This example calls a script that sets the sound level of the Macintosh to 5:

```
SetSoundLevel 5
```

Scripts that return values (acting as functions) work just like the built-in REALbasic functions. This script gets the current sound level and assigns it to a variable:

```
Dim level as Integer
level=GetSoundLevel()
```

Removing an AppleScript

To remove a script from a project, click on the script in the Project Editor to select it then press the Delete key or Control-click on the Applescript and choose Delete from the contextual menu.

Communicating with AppleEvents

AppleEvents is the core communications system between applications on the Macintosh. As a matter of fact, when you are calling AppleScript code, AppleScript is actually performing all of its magic with AppleEvents. When you choose Special ▶ Restart in the Mac OS "classic" Finder (▶ Restart on Mac OS X), the Finder sends a "Quit" AppleEvent to any open applications. This particular AppleEvent is one that all applications are required to support.

You can perform some very fast and powerful actions with AppleEvents. You create AppleEvent objects in REALbasic using the NewAppleEvent function.
AppleEvents have three parts: an event class, an event ID, and the creator code of the target application.
The Event Class and Event ID together uniquely define a particular AppleEvent. The EventClass acts as a category for logically grouping events together. While there are many standard (and even some required) AppleEvents, many applications have several custom AppleEvents for performing actions specific to the application. Consult the application's documentation or its author to get information on what custom AppleEvents may be available.

Sending AppleEvents

Once you create the AppleEvent object and populate the necessary parameters with data, you then send the AppleEvent to the target application using the AppleEvent object's Send method.

In this example, an AppleEvent is created to tell the Finder to restart the Macintosh. "FNDR" is the class of AppleEvent and "rest" is the event ID. "rest" is clearly short for "restart". Finally, "MACS" is the creator code for the Finder. The AppleEvent class has a Send method. This method is a function that returns True if the AppleEvent is successful and False if it fails.

```
dim ae as AppleEvent
ae=newAppleEvent("FNDR","rest","MACS")
if not ae.send then
  MsgBox "The computer couldn't be restarted."
end if
```

Receiving AppleEvents

In order to receive AppleEvents your project must have a subclass that has Application as its Super property value. That's because the Application class is the only class with a HandleAppleEvent event handler. When your application receives an AppleEvent, the application class HandleAppleEvent event handler is executed and the AppleEvent is passed as a parameter to the event handler.

This event handler, when called, is passed an AppleEvent object, the event class, and the event id. There are required AppleEvents that your application should support. One of the them is the Quit AppleEvent.

In this example, if the HandleAppleEvent event handler receives a quit AppleEvent from the Finder, it calls the Quit method.

```
Function HandleAppleEvent(Event as AppleEvent, eventClass as String,
    EventID as String) as Boolean
 if eventClass="aevt" and eventID="quit" then
  //the Finder wants the app to quit
  beep
  msgBox "I must quit now."
  quit
 end if
```

You can create your own set of AppleEvent classes and event IDs for your application that represent various actions your application can take in response to them.

If your are application needs to know if the Mac has gone to sleep (or more specifically, when the Mac wakes up from sleep) you can get this information very easily. It turns out that the Mac OS sends a "wake" AppleEvent to all applications when the computer wakes up from sleep.

You will need to add a class based on Application (if you don't already have one) to your project.

The HandleAppleEvent event handler of the Application class fires anytime your application receives an AppleEvent. This event handler is passed the AppleEvent, the EventClass, and the EventID. To determine whether you have received the wake event, check to see if the EventClass is "pmgt" and the eventID is "wake" in the HandleAppleEvent event handler. If you receive an AppleEvent with these values, the Macintosh has just woke up from sleep.

Keep in mind that AppleEvent classes and IDs are case-sensitive. Also, you won't receive these events when running your application in the IDE. You will receive them only in your built applications.

Sophisticated AppleEvents

AppleEvents can actually contain quite a bit of very specific data. AppleEvents for example, can be used write to applications that process data for web servers. For more information on AppleEvents, see the AppleEvent class in the *Language Reference*.

Using and Writing REALbasic Plug-ins

Many applications have their own plug-in format. Netscape Navigator, Adobe PhotoShop, 4th Dimension, are just a few examples of applications that have a plug-in format. Plug-ins are a way for an application to be extended by other programmers. For example, there is a plug-in for Netscape Navigator that allows it to play QuickTime movies that have been embedded into web pages.

REALbasic also has its own plug-in format. Plug-ins are written in languages like C and C++. For example, James Milne of Essence Software wrote a plug-in for REALbasic that plays a particular type of music file. REALbasic also uses plug-ins to manage connectivity to database back ends. You can add support for other database engines simply by writing (or obtaining from a third-party) the plug-in for that database engine.

If your application is being designed to run on Mac OS X, plug-ins must be carbonized.

Loading Plug-ins

Loading plug-ins is easy. Simply create a folder called "Plugins" in the same folder that contains REALbasic. Then drop your plug-in files into that folder. Any plug-ins in this folder will automatically be available to your projects.

Using Plug-ins

Some plug-ins are in the form of controls similar to those that appear in the REALbasic Controls list. When you have this type of plug-in in your Plug-ins folder, a new control will appear in the Controls list. Plug-in controls appear below the tools in the standard Controls list.

You use a plug-in control the same way you use any other control in the Controls list, by dragging it to a window. The Properties pane will then display any properties that can be set from the Design environment.

Plug-ins can also be a set of methods that has no interface whatsoever. Plug-ins of this type do not appear anywhere in the interface. You must have some documentation to know which methods exist in the plug-in, what the methods do, and how to use them.

Including Plug-ins in Your Stand-Alone Applications

When you build a stand-alone application from your project, any plug-ins you are using in your project will automatically be built-in to the stand-alone application.

Writing Your Own Plug-ins

If you know C or C++, you can write REALbasic native plug-ins. The REALbasic Plug-in Software Development Kit (SDK) is available on the REALbasic CD and at the REAL Software web and ftp sites. This kit contains all the information you need to write plug-ins including sample plug-ins and include files for Metrowerks CodeWarrior.

Using PowerPC Shared Libraries

PowerPC shared libraries are files that have subroutines that can be called and passed parameters. These parameters are referred to as "entry points."

In REALbasic 2005, shared and dynamically linked libraries are accessed via the Declare statement in the language. See the Language Reference for information on the Declare statement.

Microsoft Office Automation

REALbasic includes four classes that enable you to program Microsoft Office applications directly from REALbasic. On Macintosh, Office 98 and Office 2001 are supported. On Windows, Office 2000 and Office 2003 is supported, and on Mac OS X, Office X is supported. Obviously, Microsoft Office must also be installed on the machine that will run the built application.

The five classes are as follows:

Class	Description
ExcelApplication	Inherits from OLEObject and is used to automate Excel.
PowerPointApplication	Inherits from OLEObject and is used to automate PowerPoint.
WordApplication	Inherits from OLEObject and is used to automate Word.
Office	Contains all the enums you need for Office Automation.

Under Mac OS X, the libraries are installed privately within Office X. You must compile your application and copy it to the Office folder inside Microsoft Office X's parent folder. For debugging, you need to copy the REALbasic IDE inside the Office folder. This restriction may be removed.

Office Automation is not supported under Mac OS 8-9 or 'classic'.

Office Automation is the Component Object Model (COM) technology that makes Microsoft Office applications programmable.

The language that you use to automate Microsoft Office applications is documented by Microsoft and numerous third-party books on Visual Basic for Applications (VBA). Microsoft Office applications provide online help for VBA. To access the online help, choose Macros from the Tools Menu of your MS Office application, and then choose Visual Basic Editor from the Macros submenu. When the Visual Basic editor appears, choose Microsoft Visual Basic Help from the Help menu. The help is contextual in the sense that it provides information on automating the Office application from which you launched the Visual Basic editor.

The help files for VBA may not have been installed by default. If VBA Help does not appear, you will need to perform an optional install before you can access the help files. On Windows Office 2003, Office prompts you to install the VBA Help files when you first request VBA help by following the procedure described in the previous paragraph. On Macintosh, Office v.X does not install the VBA help files as part of the full install. If VBA Help does not appear, quit out of Office and locate

your master CD. Open the "Value Pack" folder and double-click the Value Pack installer. In the Value Pack installer dialog, scroll down to the Programmability topic, select it, and click Continue. The installer will then add the VBA Help files and examples to your Office installation. When the install finishes, the VBA help files will be available to the Visual Basic editor within all your Office X applications.

Microsoft has additional information on VBA at http://msdn.microsoft.com/vbasic/ and have published their own language references on VBA. One of several third-party books on VBA is "VB & VBA in a Nutshell: The Language" by Paul Lomax (ISBN: 1-56592-358-8).

Uses of Office Automation

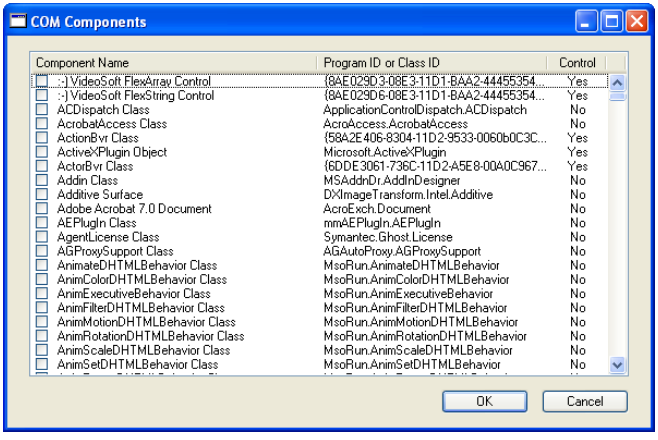
With Office Automation, you can create methods to automatically create and format documents in Word. You can use Automation to run other applications from within a REALbasic application. For example, a REALbasic application can run a hidden instance of Excel to perform mathematical and analytical operations on data in a REALdatabase. Or, you could use Word to create mail-merge letters using data in a REALdatabase.

For some examples, see the entries for the ExcelApplication, WordApplication, and PowerPointApplication classes in the Language Reference.

ActiveX Components

ActiveX components are standard user interface elements or programmable objects that enable you to build a highly customized interface rapidly. With the Windows version of REALbasic, you can add ActiveX controls and components to your application using the Project ▶ Add ▶ ActiveX Component menu item. It presents a dialog box with the list of ActiveX controls and components that are installed on your PC. That is, the list will differ depending on what is currently installed and available and would not necessarily agree with what is installed on your users' computers.

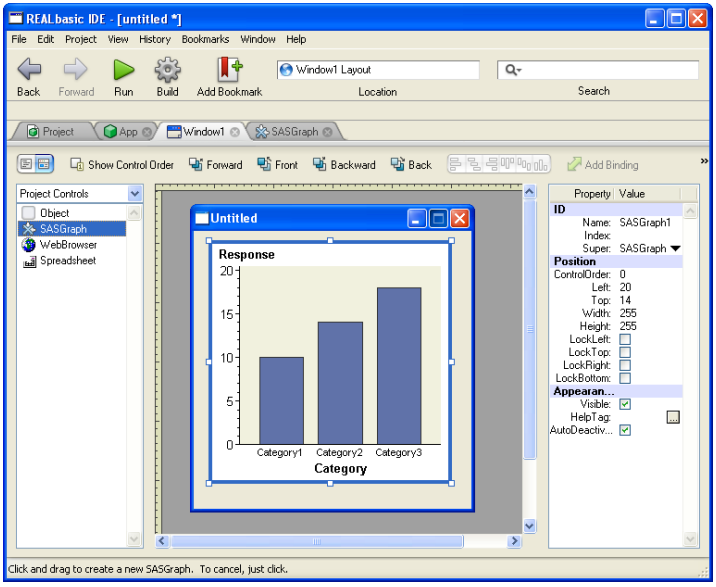
Figure 381. The ActiveX Components dialog box.



Click the CheckBox corresponding to each ActiveX component that you want to add and click OK. If you added any ActiveX controls, they will appear in the Project Controls list in the project's Window Editors and your Project Editor. If an ActiveX component is not a control, it will appear only in the Project Editor. Once added, you can incorporate the control or component into your REALbasic application as if it were a standard REALbasic object. You work with it just as if it were a REALbasic control or class.

For example, if the machine has a licensed copy of SAS Graph installed (http://www.sas.com) you can add programmatic statistical graphing capabilities to REALbasic via the SAS Graph control, as shown below.

Figure 382. The SAS Graph control added to a REALbasic window.



For information about programming ActiveX components and specific Microsoft ActiveX components, refer to Microsoft documentation in the MSDN library at:

http://msdn.microsoft.com/library/

CHAPTER 14

Building Stand-Alone Applications

When you are ready to turn your project into a standalone application, there are a few things you will need to know. This chapter will help you understand what finishing touches your application may need to make it complete.

REALbasic can create two types of standalone applications: fully-functional and demo. The fully-functional version is the same as the application that runs when you click the Run button in the Toolbar or choose Project ▶ Run in the IDE, except that no copy of REALbasic is needed (hence the name "standalone"). The demo version is the same as the fully-functional version, except that it quits automatically after five minutes.

The Professional version of REALbasic builds fully-functional standalone applications for the Windows, Macintosh, and Linux platforms. The Standard version allows you to build fully-functional applications only for the platform on which REALbasic is running. It builds demo versions for each platform on which REALbasic is *not* running.

Contents

Building Your Application

Building a standalone version of your project as an application couldn't be easier. Just click the Build button in the Toolbar or choose Project ▶ Build Application (Ctrl+B on Windows and Linux or ⌘-B on Macintosh).

By default, REALbasic will build a standalone application for the operating system you are currently running. The default name of the standalone application is "My Application" or "My Application (Mac OS X)" and will be located in the same folder as your project.

Figure 383. A standalone application (Linux, Windows and Macintosh) that use the default names and icon.

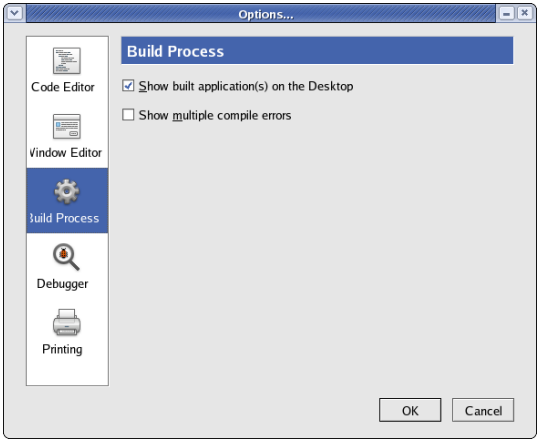
(Different Linux distributions may have different default icons.)

A standalone application includes a library of routines that support most of the controls, classes, global methods, etc. that comprise REALbasic, plus any plug-ins that are used in your application. This means, for example, that a standalone application contains code to manage ListBoxes even if your application doesn't use any ListBoxes.

When REALbasic is finished building the application, it will bring the window containing the application to the front for you. You can double-click the standalone application to start the built application.

If you don't want REALbasic to bring the window containing the application to the front, you can deselect this preference in the Build Process screen of the Options dialog box (Preferences on Mac OS X). Choose Edit ▶ Options on Windows and Linux or REALbasic ▶ Preferences on Macintosh and click the Build Process icon in the Browser area. This panel is shown in Figure 384 on page 549.

Figure 384. The Build Process options panel.



Deselect the "Show Built Applications on Desktop" preference to disable this feature. On Windows, the desktop is called "Windows Explorer" and on Macintosh, it is called the "Finder."

When you build an application, REALbasic does not ask before overwriting an existing build. If you want to keep an earlier build around, move it into another folder before rebuilding the application.

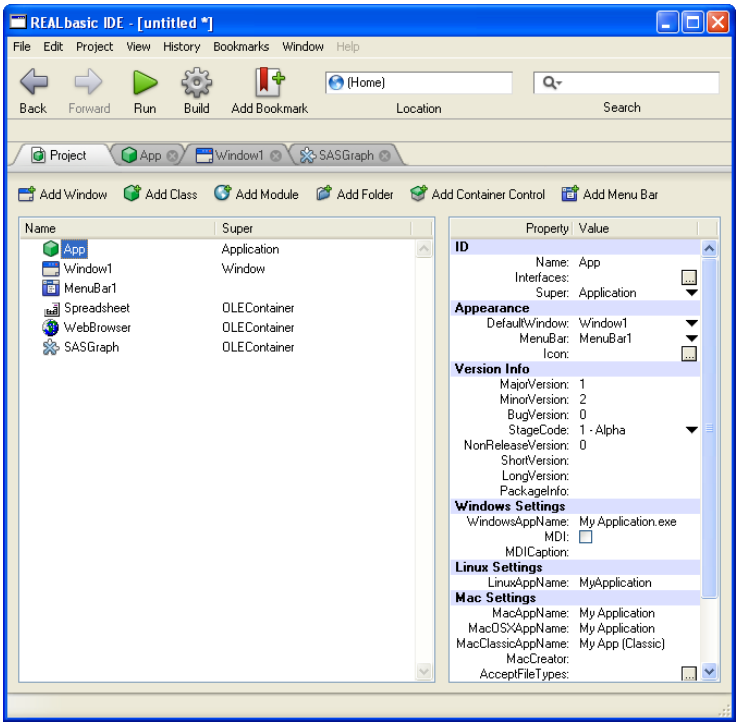
Customizing the Standalone Application's Properties

Of course, you will eventually want REALbasic to use a name you've selected for the project rather than the default name, and you may also want to build for other platforms, set version information, and platform-specific options.

You choose the target platforms for the build, the default language, and the default region in the Build Settings dialog box.

You customize your build options by setting the properties of the App class. The Properties pane for the App class contains themes for the application's Appearance, Windows, Macintosh, and Linux build settings, and Advanced settings.

Figure 385. The properties of the App class.



After you're finished entering these properties, REALbasic will use these settings when you click the Build button.

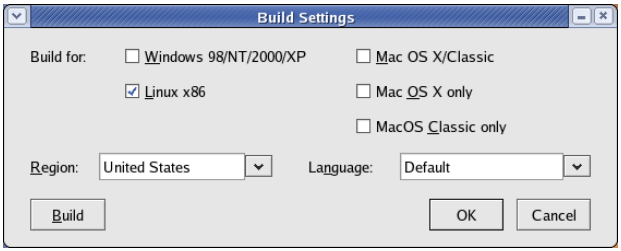
Build Settings

To customize your build settings, do this:

1 **Choose Project ▸ Build Settings.**

The Build Settings dialog box appears.

Figure 386. The Build Settings dialog box.



The CheckBoxes above the pop-up menus allow you to choose the target operating system for the build. You can build for any number of target operating systems simultaneously. By default, REALbasic builds only for the operating system that you are currently running. Your choices are:

- **Windows 98 through XP:** Builds a Windows application that will run on Windows 98/ME, NT, 2000, and XP.
- **Linux/x86:** Builds a Linux application that runs on x86-based machines. Linux builds of REALbasic desktop applications require GTK+ 2.0 or above. For information on GTK, see http://www.gtk.org. Linux builds of console applications do not require GTK.
- **Mac OS X only:** Builds for Mac OS X using the Mach-O format. This format is for the Mach kernel on which OS X is built.
- **Mac OS X/Classic:** Builds for Mac OS X and Mac OS 8-9 with CarbonLib installed using the PEF format.
- **Mac Classic Only:** Builds for the "classic" Macintosh OS only (PPC only and for versions 8-9 only). The final version of REALbasic that built for the 68K Macintosh was 3.5.2.

  REALbasic can build Mac OS X applications using either of two formats, PEF and Mach-O. PEF applications can run under both Mac OS X and on Mac OS 8-9 with CarbonLib installed. Mach-O applications run only under Mac OS X.

  The PEF structure is the format used to store programs in the Code Fragment Manager-based runtime architecture. Mach is the Unix kernel on which Mac OS X is based and Mach-O is the object file format for Mach. Mach-O is the native executable format for Mac OS X and is Apple Computer's preferred format. Console applications can be built only using the Mach-O format, but desktop applications can be built with either format.

Default Region and Language

If you have provided support for more than one language via constants (see "Using Constants to Localize your Application" on page 307), you can choose the default language for the build from the Language pop-up menu. The Region pop-up menu offers a list of major countries. Select the default region for the application as well.

Application Properties

To customize your application's properties, do this:

1. **If the Project Editor is not displayed, click its tab in the IDE.**
2. **Select the App object in the Project Editor.**

   The Properties pane changes to show the Application object's properties, shown in Figure 385 on page 550. The properties are organized into five groups: ID, Appearance, Version Information, Windows, Linux, and Macintosh settings, and Advanced settings.

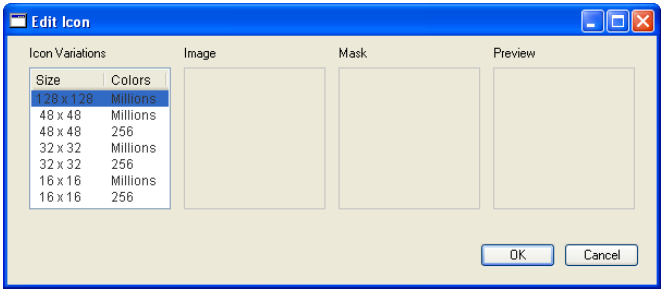
Appearance Settings

In the Appearance group, you choose the default window and menubar and install the application's icon. The default window and menubar are shown when the application starts. The drop-down lists offer all the project's windows and menubars. The default selections are the window and menubar that are added to a Desktop Application project when you first created the project with the File ▶ New Project menu command or started the REALbasic IDE application.

You can also paste in or import the custom icon for the application. If no custom application is provided, REALbasic will use the generic application icon for each platform on which the application is built. Custom icons are recognized only if the application has its own Creator code. Click the "..." button for the Icon to display the Edit Icon dialog box for the application.

REALbasic does not include an icon editor; you must design your icons in an icon editor program or image creation application and paste them into REALbasic.

Figure 387. The Edit Icon dialog box.



The Edit Icon dialog box allows you to install icons of various depths and sizes. For information on adding icons, see the section "Adding a Document Icon" on page 376. You need to provide at least the following items:

- For Mac OSX, a 128 x 128 image, and for other platforms smaller sized icon images. Even for Mac OS X it is best to provide smaller sized image files at 32 x 32 and 16 x 16.
- A mask that defines the icon's edges so that the operating systems can determine which regions in the square image are clickable.

If you are going to distribute your application on Macintosh, we highly recommend that you set a Creator code and register it with Apple Computer.

Another way of adding custom icons to your built application is to put your icon data (including 'icns' 32-bit icons) in a Resources file that you add to your project. These icons will overwrite the icon resources supplied by REALbasic itself. However, it is easier to accomplish the same thing using the Edit Icon dialog box.

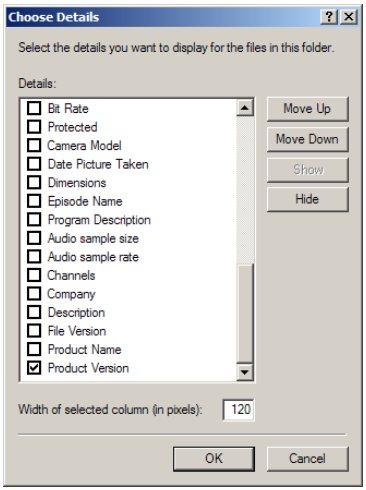
Version Information

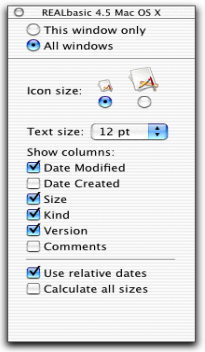
Some of the information in the Version Information area will be displayed when the user clicks on your application icon and chooses File ▶ Get Info (⌘-I) on Macintosh. The text you enter in the Package Info area appears directly below the application's name. The text you enter in the Long Version entry area appears in the Version area below the modification date.

The Version number will appear in the Get Info dialog box, the Finder's List view, and written to the 'vers' resource of your application.

The version information can be displayed on Windows if you select the "Details" view option and right+click on the column headings to display the Choose Details dialog (Figure 388). Scroll down and select the Product Version option. On Mac OS X, the comparable option is available using the View ▶ Show View Options menu command. If you are using a List view in a window, this command displays the dialog box shown on the right in Figure 388.

Figure 388. The View Options dialog box for List Views (Windows and Mac OS X).

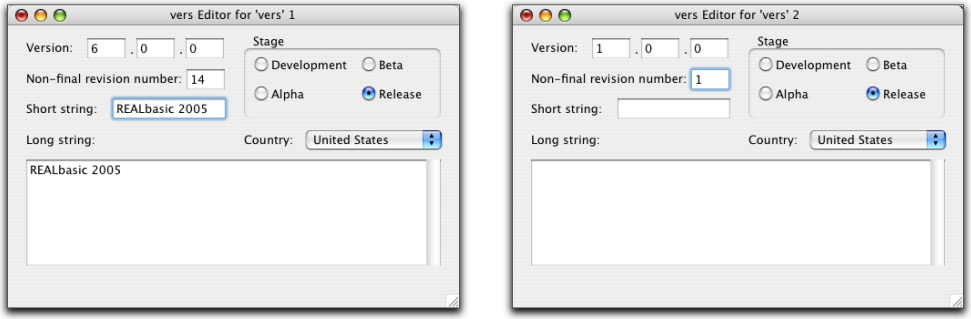




The remaining Get Info settings are not visible but are stored in your built application's 'vers' resource. The 'vers' resource can be used to store information about a individual application or, if it is part of a set of files, to the group of files. The 'vers' resource with an ID of 1 specifies version information for the application; the version resource with an ID of 2 specifies version information for a set of files.

Figure 389 shows the relationship between the Version Information settings and the 'vers' resource settings:

Figure 389. Macintosh 'vers' resources 1 and 2.



The string in the Long Version area for vers ID2 corresponds to the Package Info field in the App class's properties.

The following elements are stored:

- Major version level, in binary-coded decimal format. Accessible only from the 'vers' resource in the first byte.
- Minor version level, in binary-coded decimal format. Accessible only from the 'vers' resource in the second byte.
- Stagecode. The levels are coded as follows:

  Table 40: Values corresponding to Stagecode.

Value	Description
0	Development
1	Alpha
2	Beta
3	Release

  This information appears as the Release level in Windows Properties.
- Pre-release version level. Accessible only from the 'vers' resource.
- Region code. Identifies the script system for which the application is intended. The values stored in the 'vers' resource are given in Table 41 on page 564.
- Short version. Identifies the version number of the software. This may be displayed, at the end-user's option, in the Finder's List views.
- Long version/Package Info. Long Version contains the version number and other identifying information about the company, developer, or group of files. For 'vers' ID 1, the Long Version is displayed in the version field of the built application's Get Info window. For 'vers' ID 2, the string is displayed under the application's name

and next to the application's icon at the top of the Get Info window. It is referred to as 'Package Info' in the Build Application dialog box.

You can find more information about the 'vers' resource at http://developer.apple.com/techpubs/mac/Toolbox/Toolbox-454.html.

Using the GetResource method of the ResourceFork class, you can access the information contained in the 'vers' resource of your built application.

Windows Settings

In the Windows Settings group you give the Windows build of your application a name and, if desired, choose to run it as a Multiple Document Interface (MDI) application.

If you select the Multiple Document Interface option, the application's windows will be enclosed in the "parent" MDI window. If you select this option, you can enter the caption for the MDI window that appears in its title bar.

Select Multiple Document Interface if you want your Windows application to run inside a "master" window. Enter the text of the master window title in the MDICaption area.

If you deselect Multiple Document Interface, your application's windows will appear alongside other applications' windows (like the REALbasic IDE) and have their own menubars. If your application can open more than one document window, it will launch another copy of itself.

The MDIWindow class in the language allows you to set several properties of the MDI window, such as its initial location and size, minimum size, and title. For more information, see the entry for the MDIWindow class in the *Language Reference*.

There are three optional fields in the Windows Settings group:

- Internal Name: This is useful when your product has a different internal name than its external name. If you leave Internal Name blank, it will be set to the name of the application.
- Company Name: The name of the executable file. Use this if the name of the executable is different from the (public) product name that is visible in the Start menu. For example, "winword.exe" versus "Microsoft Word".
- Product Name: The name of the product as installed in the Windows Start ▶ All Programs menu.

Linux Settings

The Linux Settings group enables you to give the Linux build a name.

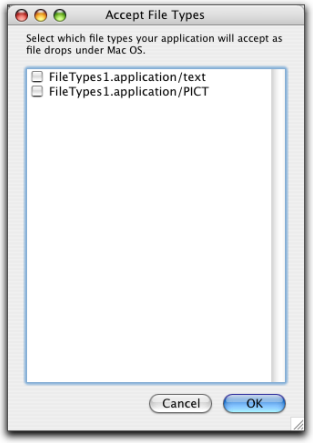
Mac Settings

Use the Mac OS Settings group to set the names of the Macintosh 8-9 (a.k.a., "classic") and Mac OS X applications, set a Creator code for the built application, and specify the file types that the application will accept via drag and drop. The Creator code can be set using a global constant (that is declared in a module) or a public constant using the syntax *modulename.constantname* or *windowname.constantname*.

File Types for Macintosh Drag and Drop

To specify the acceptable file types for drag and drop, first define the file types in a File Types Set Editor. Then click the "..." button for the AcceptFileTypes property in the App object's Properties pane. The following dialog box will appear, listing the file types that you have defined.

Figure 390. The Accept File Types dialog box.



The AcceptFileTypes dialog will list all of the currently defined File Types. It uses the syntax FileTypeSetName.FileTypeName. To indicate that the Macintosh build of the application will accept drag and drop for the file type, select the file type's CheckBox by clicking its CheckBox.

Registering Your Creator Code

Each application's creator code should be unique. This is because the Finder uses these codes to determine which application to launch when a file is double-clicked. The Finder simply locates the first application it can find with a matching creator code.

You can register your application's creator code with Apple Computer to be reasonably sure that it's unique. For more information about registering Creator codes, see the Apple developer web site at http://developer.apple.com.

Advanced Settings

The Advanced Settings group has properties for setting the minimum and suggested memory for the Mac OS Classic build. All other operating systems allocate memory dynamically. It also has an option to include function names in the debugger. Normally this option is not selected.

Memory Settings

The amount of memory reserved for your application can be set for your Mac OS "classic" build. These are the MinimumMemory and SuggestedMemory properties in the Advanced group. If you need to set a small value for these parameters, you should experiment to determine how much memory the application will need. Run the standalone application with different memory settings until you find the minimum

setting under which it runs satisfactorily. Then update the settings to reflect your results.

For example, loading lots of data into memory, especially pictures, increases the memory requirements. If you are using the sprite engine, the more sprites displayed at once, the more memory you will need. Unfortunately, there is no really straightforward logic to determining the memory requirement for your application.

Preparing your Application for Compilation

Although the process of creating a standalone application is very simple, you should take into account any special features or limitations of the target operating system so that you can optimize your application's performance.

REALbasic offers a very convenient way of testing your application under a different operating system: remote debugging. If you have a computer that is running a second target operating system, you can use remote debugging to send it a debug build of your application and launch it automatically. Remote debugging is available only in the Professional version of REALbasic.

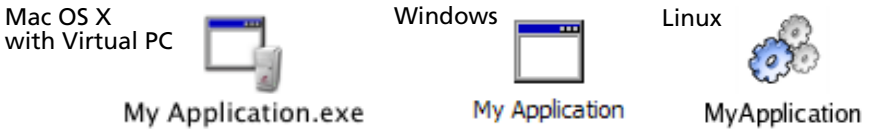
For information on how to configure your computers for remote debugging, see the section "Remote Debugging" on page 514.

This section covers some of the issues affecting the behavior of an application on each of the platforms.

Compiling for Windows

To compile for Windows, check the Windows CheckBox in the Build Settings dialog box area and enter the name of the .exe file in the Windows group in the App object's Properties pane. The compiler will create a single executable application for the Windows environment. On Window, the application will have a generic application icon unless you have provided a custom icon. On Macintosh, the Windows application will have a Virtual PC™ icon if Virtual PC is installed. You can double-click the Windows build to launch Virtual PC and open your application.

Figure 391. Windows builds of a REALbasic application.



Windows Considerations

Before building an application for deployment on Windows, you should check the following issues for compatibility:

- **End of Line Characters**: When inserting text into EditFields via code, keep in mind that Returns are followed by Line Feeds. Use the EndOfLine function rather than hardcoding end of line characters.

- **Non-ASCII Codes**: Characters above ASCII 127 are not identical on all platforms and also differ by language. For example, the bullet character on the Macintosh is 165 but on Windows it is 149 (US operating systems). When you read in text from an outside source, you can use the optional Encoding parameter of the Read, ReadLine, or ReadAll methods to specify the encoding that the incoming text uses. Internally REALbasic stores the encoding with each text string. If you need to write text out to an application or platform that expects a particular encoding, you can call the ConvertEncoding method to convert the text to the desired encoding.
- **Windows GUI**: If you are developing your application on Macintosh or Linux, it's important to consider Windows user interface guidelines. Otherwise, your application will look like a port from another platform to Windows rather than a genuine Windows application.
- **Control Order**: If you are developing the application on Macintosh for deployment on other platforms you should check the control order since controls like CheckBoxes and PushButtons can get the focus on Windows and Linux. Make sure you test the control order on other platforms before building your application.
- **Multiple Document Interface:** Some MDI (multiple document interface) applications on Windows maximize the MDI frame window when launched. The MDI frame window is the parent window in which the applications windows open.

  If you are compiling a Win32 version of your REALbasic project and would like to maximize the MDI frame window when your application launches, you can call the Maximize method of the MDIWindow class in the Open event in the App object. The the App object is added to your Project Editor automatically. Be sure to specify that you want the application to run in an MDI window in the Windows settings group of the App class's properties.
- **Mac-only Controls**: Some controls (like the StandardToolbar item) are unique to the Macintosh and don't have a Windows or Linux counterpart. The Toolbar item and Standard Toolbar item controls are for Mac OS X 10.2 and above; if you use them, you will need to provide alternate interfaces for all other platforms that you support.
- **Windows Menus**: The text of certain standard menu items on Windows are different. For example, "Exit" is used instead of "Quit". You can use the REALbasic constants system to localize your application's interface for Windows in this respect (see "Using Constants to Localize your Application" on page 307). The Exit or Quit menu in REALbasic is already handled via constants in the App class.
- **Mac-only Features**: A few REALbasic capabilities are Macintosh-specific. For example, AppleEvents and AppleScript, the ToolbarItem and StandardToolbarItem controls, and the DockItem class. Another example is the Permissions class, which

is for Linux and Mac OS X only. You can use conditional compilation to isolate this code. It uses the structure:

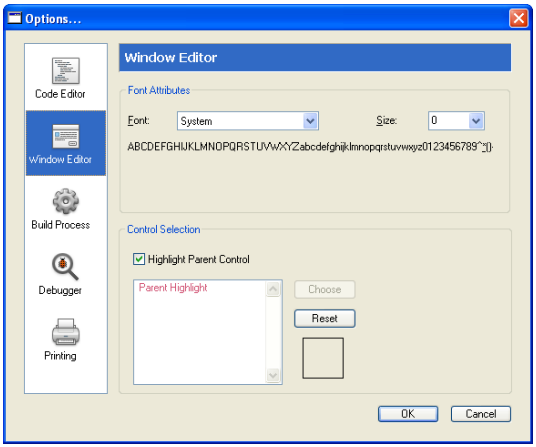
```
#If TargetBoolean then
 //platform-specific code, included in
 //the built app when TargetBoolean is True
#Elseif TargetBoolean2 then
    //platform-specific code for TargetBoolean2
#Endif
```

TargetBoolean is a boolean constant or constant expression that lets you selectively include code that will be included only in a particular build. You can use the built-in boolean target compiler constants TargetMacOS, TargetMachO, TargetMacOSClassic, TargetHasGUI, TargetCarbon, TargetLinux, or TargetWin32. These constants test *which type of code is compiling*. You can also use constant expressions that evaluate to True or False. For example, you can use the RBVersion constant to determine whether the user is using a particular version of REALbasic or at least a minimum version level, e.g., version 4.5 and above.

See the section in the *Language Reference* on Cross-Platform Development and the descriptions of the conditional compilation constants for more information.

- **Platform-specific FolderItems:** Some of the functions that return references to Mac OS-specific or Linux-specific folders will return Nil on Windows and vice versa. See the entry for SpecialFolders in the Language Reference and access special OS FolderItems via this module.
- **API Calls**: With the Declare statement, you can make API calls for Macintosh, Linux, or Windows platforms. Of course, the nature of the call will differ by platform. Use conditional compilation to isolate both the Declare statements and your usage of your toolbox calls later in your code.
- **Font sizes for controls:** The three target platforms supported by REALbasic have their own conventions regarding default system fonts and font sizes. Often, a font size that looks good on one platform is too large or too small on another platform. For that reason, the REALbasic Options dialog box has an option for choosing the default font size for the target platform to use as the default font size for all controls that use text. This, in effect, allows you to choose more than one default font size for controls simultaneously. Choose a font size of zero to use this option.

Figure 392. Choosing the default font size for the target platform for controls.



You can also set the TextSize property of any control to zero to invoke this option for the particular control (if the option is not selected globally, as shown in Figure 392). You can also specify the System font or the SmallSystem font rather than a specific font to instruct REALbasic to use the system (or small system) font for the target platform.

Mac OS X Considerations

Before building your application for Mac OS X, you should carefully test all the interface elements to make sure that they look right (i.e., buttons, Popup menus, ListBoxes, etc. are properly sized and look good with the fonts used in Mac OS X). Also any shared libraries that the application uses must be carbonized.

If you are a Windows or Linux user and are cross-compiling for Mac OS X, you should familiarize yourself with the Apple's Human Interface Guidelines. It is a detailed specification for Apple-recommended user interface design. The complete user interface guidelines are at:

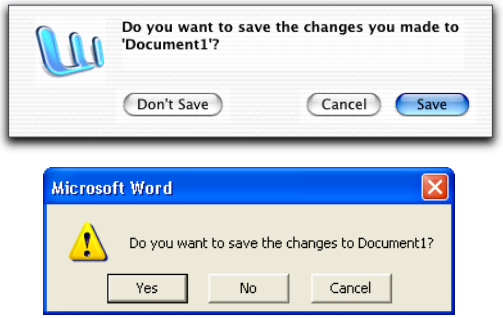
http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/

The following are some of the major differences between Apple interface conventions and those of other platforms. Some differences can be handled via careful interface design while others may require alternate versions of windows, dialogs, and other interface elements. To manage alternate versions of objects, you can use the #If statement to conditionally compile for different target platforms. See the *Language Reference* for information about #If and the target platform constants used for conditional compilation.

- Macintosh uses a single menubar which is always at the top of the screen. Individual windows do not have menubars. Although REALbasic supports multiple menubars that can be associated with different windows, the use of this feature is contrary to Apple user interface guidelines.

- The Exit menu item on Windows is named "Quit" on Macintosh. You can use the REALbasic constants system to localize your application's interface for Windows in this respect (see "Using Constants to Localize your Application" on page 307). This is done within REALbasic, for example.
- An application's preferences menu item is located in the application's own menu, rather than under the File, Edit, or Tools menu. REALbasic has a special class for handling this issue, the PrefsMenuItem class. Use this class rather than the MenuItem class for your Preferences menu item.
- Macintosh does not support Windows' Multiple Document Interface. This makes windows document-centric rather than application-centric because a document window is never enclosed in a parent window that identifies the application and holds the application's menubar. Moreover, each Mac OS X window exists in its own layer in the Finder. That is, clicking on one of an application's windows does not bring all of the application's windows to the front. As a result, a Macintosh user can interleave one application's windows with other application's windows. If you designed your application as an MDI application, be sure to check out its behavior in a non-MDI environment.
- The standard Macintosh mouse is a one-button mouse, although third-party multiple-button mice are available and supported by the Mac OS. However, you can't expect that all your users will have a third-party mouse. On Macintosh, the equivalent of the Windows' right+click gesture is Control-click. Be sure that you application does not rely on mouse gestures that are not supported by the one-button Apple mouse.
- Mac OS X uses the "Dock" as an application and document launcher. REALbasic's DockItem property of the Application class enables you to control the appearance of your application's Dock icon and the DockItem property of the Window class enables you to control the appearance of the icon representing one of your application's documents (When a Mac OS X user minimizes a document window, it appears as an item in the Dock). These icons should be designed as 128 x 128 pixel icons.
- In dialog boxes, PushButton captions use verbs rather than "Yes" and "No" and are arranged differently. Typically, the positions of the "validate" and "cancel" buttons are reversed. For example, the two dialogs shown in Figure 393 are the "save changes" dialogs from Microsoft Word for Mac OS X and Windows XP. As you can see, the arrangement and wording of the PushButtons are significantly different. Also, modal dialogs such as this are implemented on Mac OS X as sheet windows when they pertain to a particular window.

Figure 393. "Save Changes" on Macintosh OS X and Windows XP.



- Toolbars that contain many, many small icons are discouraged by Apple. Apple favors the use of a few "high quality, larger" icons in the toolbar which are labeled with text below the icon rather than via a hidden "tips" window. REALbasic's ToolbarItem and StandardToolbarItem classes can be used to create such toolbars. Mac OS X applications also use the Drawer window to reveal additional choices or options when a toolbar icon is clicked. Apple also recommends the use of floating palettes as an alternative to extensive toolbars.
- There is a standard list of reserved keyboard shortcuts for Macintosh. Your application should not try to override them. Aqua user interface guidelines also specify certain recommended keyboard shortcuts for common operations, such as the keyboard equivalents for the standard File and Edit menu items. These keyboard shortcuts are listed in the Aqua Human Interface Guidelines, available at the URL given above.
- Mac OS X uses filename extensions, as does Windows, but Mac OS "classic" uses Type and Creator codes to code file attributes. If users move documents created by your application to Mac OS "classic" you need to create and register Type and Creator codes for your application.
- Carbon applications include an automatically generated 'plst' resource which describes your application to the Mac OS X Finder. If you wish, you can override this by including your own 'plst' 0 resource in a resource file in the project.

Linux Considerations

If you are building your application for Linux, from another platform, you should carefully check out the appearance of the Linux version of your interface and adjust the font size and size of controls for maximum readability.

You should keep in mind the following information.

Requirements

Linux applications run on only on x86 machines and REALbasic desktop applications require GTK+ 2.0 or above (which has its own requirements, such as GDK, Pango, Atk, etc.). Depending on your distribution of Linux, you may not already have this pre-installed. However, you can download and install the GTK+ 2.0

libraries from http://www.gtk.org. You may also find pre-built libraries from your Linux distributor's home page or GTK may be included on your distribution CD as an optional install item.

General Information

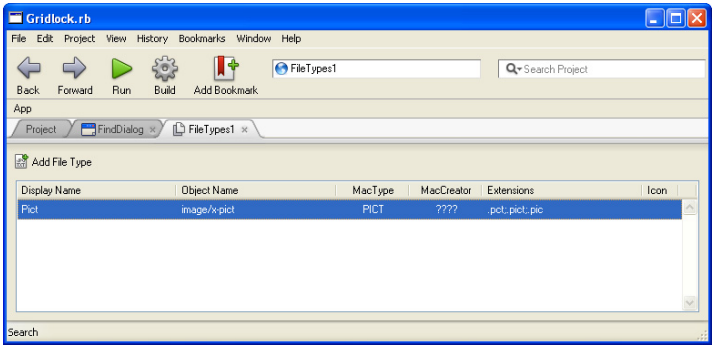
Some controls and operations that work on Macintosh and/or Windows have limitations under Linux. This is detailed below.

- **Rb3DSpace:** The Rb3DSpace control requires OpenGL.
- **Sounds:** Requires libsndfile (http://www.zip.com.au/~erikd/libsndfile/) but is weak linked to your app, so your app should still run even if the end user doesn't have this library installed, but he or she can't play any sounds.
- **Canvas:** Scroll will redraw the Canvas's contents if there are certain controls on top of the Canvas, such as a PushButton.
- **Drag and Drop:** Dragging over controls that have mouse over effects, such as PushButtons, do not update the drag bounds properly.
- **EditField:** The SelCondense, SelExtend, SelOutline, and SelShadow styles are not supported. These are nonstandard text styles that were introduced with Mac OS classic.
- **FolderItem:** Setting the creation date of a file or folder is not supported.
- **Menus:** A window cannot be assigned the same menubar as another. If you want two windows to have the same menus you need to duplicate the menubar.
- **MoviePlayer** and **QuickTime** related classes are not supported.
- **PushButton**, **CheckBox**, and **RadioButton:** You can't suppress the depressed state of buttons, CheckBoxes, and RadioButtons when returning True from the MouseDown event.
- **ScrollBars:** ScrollBars behave a little bit differently than on Macintosh and Windows. Clicking on the arrow buttons without LiveScroll does not trigger the ValueChanged event until you release the mouse button.
- **Speak:** The Speak method is not supported in the initial release.
- **RadioButtons:** In a group of RadioButtons, one must always be selected.
- **PopupMenus:** There is no support for Separators.
- **NotePlayer:** The NotePlayer control is not supported.
- **Printing:** Printing requires libgnomeprint 2.2 (or above) and CUPS installed. Printing is supported only to PostScript printers. On Linux, there is no Page Setup dialog. Calling the PrinterSetup method returns False and no dialog is shown to the user.

Assigning Custom Document Icons

As you already know, you can assign a custom application icon in the App class's properties pane. If your application creates documents, you will probably want to assign icons that match the theme of your application icon, to the documents it creates. This can be done in the File Type Sets Editor. To add the appropriate document icons, click the Icon button in the File Types Sets Editor (Figure 394) to display the custom file type dialog box. For more information, see the section "Adding a Document Icon" on page 376.

Figure 394. The File Types Set Editor.



The Thread Manager

If you are building an application for Mac OS "classic," your application will not require the Thread Manager unless you are creating subclasses based on the Thread class.

Region Codes

The following table gives the Region Codes that are used in the 'vers' resource for built applications.

Table 41: Region codes used in the 'vers' resource.

Code	Value	Code	Value
00	US	22	Malta
01	France	23	Cyprus
02	Britain	24	Turkey
03	Germany	25	Yugoslavia
04	Italy	33	India
05	Netherlands	34	Pakistan
06	Belgium-Lux.	36	It. Swiss
07	Sweden	40	Anc. Greek
08	Spain	41	Lithuania

Table 41: Region codes used in the 'vers' resource.

Code	Value	Code	Value
09	Denmark	42	Poland
10	Portugal	43	Hungary
11	Fr. Canada	44	Estonia
12	Norway	45	Latvia
13	Israel	46	Lapland
14	Japan	47	Faeroe Isl.
15	Australia	48	Iran
16	Arabia	49	Russia
17	Finland	50	Ireland
18	Fr. Swiss	51	Korea
19	Gr. Swiss	52	China
20	Greece	53	Taiwan
21	Iceland	54	Thailand

CHAPTER 15

Converting Visual Basic Projects to REALbasic

Because of the similarities between REALbasic and Visual Basic, creating a Macintosh version of a Visual Basic application is fairly easy. REALbasic can save you hours of time by handling the tedious job of recreating the interface and pasting in your code into all the various event handlers and methods.

Contents

Importing Forms and Code

REALbasic can import Visual Basic 2.0 (or greater) forms and the code associated with them. REALbasic recreates the interface and imports the code for all of the various controls into the appropriate event handlers and methods.

To import VB forms into a REALbasic project, do this:

1. **(Optional, for Macintosh users) Move your Visual Basic form files (those ending in.frm) over to your Macintosh.**
2. **Open a new or existing project in REALbasic.**
3. **Drag the form files from the desktop into the Project Editor to import them.**

Because there are differences between REALbasic and Visual Basic, there are going to be errors in your code that you will need to correct. Once you finish importing the form files, you can run your project to begin tracking down those errors.

Making The Conversion Easier

There are a number of steps you can take proactively to make the process of converting a VB project to REALbasic easier. REAL Software ships a utility, VB Project Converter (VBPC) that simplifies the process. To convert a project, drag and drop the VB project onto the VB Project Converter application. It will automatically add all the files that belong to the project. All project files must be in the same location as the project file. Form resource files (*.frx) must be in the same location as its form. You do not have to add *.frx form resource files; these will be read when needed by the VB Project converter.

You can deselect any file if you do not want to include it in the REALbasic project.

Known Issues

The following issues deal with differences between the architectures of VB and REALbasic.

- **Controls:** If a control is not supported in REALbasic, VBPC will convert it to a Canvas control. VB controls that serve as containers of other controls or views may produce a corrupted REALbasic project since this version of VBPC does not parse inside elements.

  VBPC provides Canvas controls for a few controls not supported in REALbasic. A ckMonthView Canvas control will be added to your project when a MonthView control is found. ckMonthView offers the basic appearance and functionality of the VB MonthView control.

- **Menus:** Menus are supported only in forms. REALbasic supports only Option and Shift besides the standard menu shortcut. Shortcuts using function keys will be replaced by the key number. The Delete key will be replaced with a ‘D’, the Insert key with ‘I,’ and Backspace with ‘B’. Any other special key will be removed.

- **MDI**: This release of VBPC offers no support for MDI projects. MDI support is underway. MDI support in REALbasic is done by means of an MDI class and therefore is not a container. You will not be able to have controls in the MDI parent window in REALbasic.
- **Icons**: An Icon is imported as an image containing a mask and the actual image.
- **Constants**: VBPC will attempt to guess the data type of a constant. You must check the data type of any constant once you open the converted project in REALbasic.
- **Database**: Database support is limited in this version of VBPC. ADAO and DAO data controls will be added to your project. Since an actual database is not added to your project, bound controls such as EditFields will have their DataField property set but not their DataSource property. To convert an MS Access database to a REAL database, check Access Converter from Jose Cuevas. Datadesigners, Crystal Reports and similar products are not supported.

What Are My Database Options?

Visual Basic applications often use Microsoft Access or the Jet database engine that comes with Visual Basic to provide single-user database capabilities. You can convert these applications to use the built-in REALbasic database engine or any other supported data source. The REALDatabase data source is included in both the Standard and Professional versions of REALbasic.

Index

B

C

D

E

F

G

P

Q

R

S

T

U

V

W

X

Z