IBM VisualAge for Java**

Getting Started

for OS/2* and for Windows**, Version 1.0

S430-4086-00

IBM VisualAge for Java**

Getting Started

for OS/2* and for Windows**, Version 1.0

S430-4086-00

Before using this information and the product it supports, be sure to read the general information under “Notices” on page v.

First Edition (July 1997)

This edition applies to Version 1.0 of the VisualAge for Java product, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term "VisualAge", as used in this publication, refers to the VisualAge for Java product set.

Order publications by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. eastern standard time (EST). The phone number is (800) 879-2755. The fax number is (800) 284-4721.

You can also order publications through your IBM representative or the IBM branch office serving you locality. Publications are not stocked at the address below.

If you have comments about this document, address them to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada, M3C 1H7

FAX: (416) 448-6161
torrcf@vnet.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices v

Trademarks vii

About this document ix
What this document includes ix
Sample programs in this document x
Who this document is for x
About this product x
Conventions used in this document x

Chapter 1. The Basics 1
What is VisualAge for Java? 1
  Rapid application development 1
  Create industrial-strength Java programs 2
  Maintain multiple editions of programs 2
Key concepts 2
  Development with a repository 2
  The workspace and the repository 2
  Importing and exporting code 3
  The Workbench 3
  Visual programming with the Visual Composition Editor 5
  Beans 5
  Connections 5
  The Visual Composition Editor 5

Chapter 2. Building your first applet 9
Introduction to building your first applet 9
Getting started with your first applet 9
  Starting VisualAge for Java 9
  Now that you are up and running 10
Using a SmartGuide 11
  Creating an applet, a project, and a package 11
Using the Visual Composition Editor 12
  Working with beans 12
Building the To-Do List applet 13
  Adding TextField and Label beans 14
  Changing the text of a Label 14
  Adding a List 14
  Adding Buttons 15
Sizing and aligning visual beans 15
  Sizing, aligning, and distributing beans 16
  Saving your work so far 17
Correcting mistakes 17
Connecting beans 17

Event-to-method and parameter-from-property connections 17
Saving and testing the To-do List applet 20
Saving your visual bean 20
Testing the applet 20
Saving your workspace 21

Chapter 3. Adding state checking to your applet 23
Introduction to adding state checking to your applet 23
Finding your To-Do List applet in the Workbench 23
Versioning an edition of your applet 24
Adding state checking to your applet 25
Desired behavior of the Remove button 25
Overview of adding the desired behavior to the Remove button 25
Open your To-Do List applet in the Visual Composition Editor 25
Set the properties of the Remove button 26
Create a new method to check if an item is selected 27
Add a connection to enable the Remove button 27
Add connections to disable the Remove button 28
Saving and testing your changes 30

Chapter 4. Creating the To-Do File program 31
Introduction to creating the To-Do File program 31
Behavior of the To-Do File program 31
Steps for creating the To-Do File program 32
Creating a new class 33
Creating a new class: creating a skeleton 33
Creating a new class: adding a method for reading files 35
Creating a new class: adding a method for writing files 38
Using the Scrapbook to test code 41
Adding buttons to the To-Do List applet user interface 42
Adding the ToDoFile class to the free-form surface 44
Adding file dialog beans to the free-form surface 45
Connecting the Open To-Do File button 47
Create the connection to show the file dialog 48
Create the connection to dispose of the file dialog 48
Create the connection to invoke readToDoFile 49
Testing the Open To-Do File button 50
Connecting the Save To-Do File button 51
Create the connection to show the file dialog 52
Create the connection to dispose of the file dialog 52
Create the connection to invoke writeToDoFile 53
Saving and testing the completed To-Do File program 55

Chapter 5. What else you can do with the Visual Composition Editor 57
Introduction to what else you can do with the Visual Composition Editor 57
Manipulating beans 57
Selecting beans 57
Selecting several beans 58

Deselecting beans 58
Moving beans 58
Copying beans 59
Copying beans using the clipboard 59
Deleting beans 59
Sizing, aligning, and positioning beans 60
Sizing beans 60
Aligning beans 60
Matching the dimensions of another bean 61
Distributing beans evenly 61
Changing bean properties 62
Opening the Properties window for a bean 62
Changing bean colors and fonts 63
Changing the color of a bean 63
Changing the font of a bean 64
Portability of colors and fonts 65
Connecting beans 65
Property-to-property connections 66
Event-to-method connections 67
Event-to-script connections 68
Parameter connections 69
Changing the properties of connections 71
Connection parameters 72
Manipulating connections 74

Chapter 6. Managing editions 79
Introduction to managing editions 79
About editions 79
Versioning an edition 81
Updating your code again 81
Creating a new edition 82
Adding a counter to the ToDoFile program 82
Returning to a previous edition 85
Exploring the Repository 85
Examining examples in the repository 87
Summary 88

Chapter 7. What else you can do 89
Introduction to what else you can do 89
Printing program elements 89
Changing the default printer 90
Navigating 90
Moving between windows 90
Windows you can open from the Window menu 91
Searching 94
Searching for a program element 94
Searching with the Search secondary window 94
Searching from the Workspace menu 96

Browsing . . . 97
Browsing a project . . . 97
Browsing a package . . . 98
Browsing a class . . . 99
Browsing an interface . . . 99
Browsing a method . . . 100
Debugging . . . 100
Setting breakpoints . . . 101
Removing breakpoints . . . 101
Using the Debugger window and the Breakpoints window . . . 102
Support for JavaBeans . . . 104
What are JavaBeans? . . . 104
Bean Features . . . 104
BeanInfo Classes . . . 105
The BeanInfo page . . . 105
Using the BeanInfo Page . . . 107
Customizing the Workspace . . . 112
Setting customization options . . . 112

Chapter 8. Accessing enterprise data . . . 117
Data Access Builder . . . 117
Thin-client model . . . 118
Generating the schema mapping . . . 119
Customizing the schema mapping . . . 121
Generating the data access beans . . . 123
Using the generated GUI . . . 125
Connecting to your own GUI . . . 126
Distributing and deploying the GUI . . . 127
CICS Access Builder . . . 127
CICS access overview . . . 128
Using the Create COMMAREA Bean SmartGuide . . . 129
The Unit of Work bean . . . 131
Connecting the CICS access beans to the user interface . . . 131
RMI Access Builder . . . 132
Overview of RMI access through proxy beans . . . 133
Using the Create Proxy Bean SmartGuide . . . 134
Creating RMI stubs and skeleton classes . . . 136
Connecting the client-side server proxy . . . 136
Starting the server . . . 137
C++ Access Builder . . . 139
Generating the C++ wrapper and stub beans . . . 139
Working with the generated makefile . . . 142
Local and remote access scenarios . . . 143
Importing the beans into the IDE . . . 143
Distributing and connecting the stub bean . . . 144

Chapter 9. More information about VisualAge for Java . . . 145
Printing material . . . 145

Index 147

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

IBM may change this publication, the product described herein, or both.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

CICS
DATABASE 2
DB2
IBM
Operating System/2
OS/2
VisualAge

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc in the United States and in other countries.

Microsoft, Windows and Windows NT are trademarks of Microsoft Corporation.

Acrobat Reader and Adobe are trademarks of Adobe Systems Incorporated.

Other product, service and company names, which may be marked with a double asterisk, may be trademarks of their respective owners.

About this document

The purpose of this document is to introduce you to:

- The basic concepts and terms you need for using VisualAge for Java
- The fundamentals you need to know in creating an application using VisualAge for Java

To help achieve these goals, we guide you through creating a simple Java applet. Then we guide you through adding features to this applet.

What this document includes

This document is divided into nine chapters:

- Chapter 1, “The Basics” on page 1 introduces the overall capabilities of VisualAge for Java and outlines the concepts you need to know.
- Chapter 2, “Building your first applet” on page 9 introduces the visual programming features of VisualAge for Java by leading you through the process of creating a simple applet.
- Chapter 3, “Adding state checking to your applet” on page 23 gives you more details on the visual programming features of VisualAge for Java and shows you how to make improvements to the applet you created in the previous part. This part also introduces VisualAge for Java's approach to code management.
- Chapter 4, “Creating the To-Do File program” on page 31 shows you how to add more features to your applet and gives you more details on VisualAge for Java's overall coding environment.
- Chapter 5, “What else you can do with the Visual Composition Editor” on page 57 gives you more details on the powerful visual programming capabilities of VisualAge for Java.
- Chapter 6, “Managing editions” on page 79 gives you more details on the edition control features of VisualAge for Java.
- Chapter 7, “What else you can do” on page 89 gives you more information on the following features of VisualAge for Java:
  - Printing
  - Navigating
  - Searching
  - Browsing
  - Debugging
  - Support for JavaBeans**
  - Customizing your programming environment

Chapter 8, “Accessing enterprise data” on page 117 provides an overview to using the robust builder tools that are shipped with VisualAge for Java Enterprise. [ENTERPRISE]

Chapter 9, “More information about VisualAge for Java” on page 145 describes the overall help system that comes with VisualAge for Java. It also gives you details on printing material from the help system.

Sample programs in this document

This document includes the following sample programs:

The "To-Do List" applet in Chapter 2, “Building your first applet” on page 9 guides you through the basic steps in creating an applet. You can find a completed version of this example in the `COM.ibm.ivj.examples.vc.todolist` package in the `IBM Java Examples` project in the VisualAge for Java repository.

The "To-Do File" program in Chapter 4, “Creating the To-Do File program” on page 31 guides you step-by-step through adding new features to the simple To-Do List applet. You can find a completed version of this example in the `COM.ibm.ivj.examples.vc.todofile` package in the `IBM Java Examples` project in the VisualAge for Java repository.

See “Examining examples in the repository” on page 87 for details on how to examine the completed versions of these examples.

Who this document is for

This document is written for programmers who want to become familiar with the basic use of VisualAge for Java, and for anyone who wants an overall perspective on the product. It introduces you to the basic concepts behind building programs using VisualAge for Java, explains the general process of visual programming with VisualAge for Java, and walks you through two sample programs. To get the most out of this document, you should be familiar with the basics of the Java language.

About this product

VisualAge for Java is a complete, integrated environment for creating Java applications and applets.

VisualAge for Java gives you interactive visual programming tools and a set of JavaBeans that represent common interface components. See “Support for JavaBeans” on page 104 for more details on JavaBeans. You create programs by assembling and connecting beans. In many cases, you may not even have to write code. When you do need to write code, VisualAge for Java provides a state-of-the-art, integrated development environment in which to do your coding.

Conventions used in this document

The following conventions are used in the text:

Highlight style	**Used for**
Boldface	New terms the first time they are used
	Items you can select, such as buttons and menu choices
Italics	Special emphasis
	Method names in general discussion. Method names that you can select in the VisualAge for Java environment, however, are boldface, and method names in code samples are monospace font.
	Property and event names
	Text that you can enter
`Monospace font`	Examples of Java code
	File names

Chapter 1. The Basics

What is VisualAge for Java?

VisualAge for Java is an integrated, visual environment that supports the complete cycle of Java program development. In particular, VisualAge for Java gives you everything you need to perform the development tasks described in this section.

Rapid application development

You can use VisualAge for Java's **visual programming features** to quickly develop Java applets and applications. In the **Visual Composition Editor** (described in “Visual programming with the Visual Composition Editor” on page 5) you point and click to:

- Design the interface for your program
- Specify the behavior of the interface elements
- Define the relationship between the interface and the rest of your program

VisualAge for Java generates the Java code to implement what you visually specify in the Visual Composition Editor. In many cases you can design and run complete programs without writing any Java code.

In addition to its visual programming features, VisualAge for Java gives you SmartGuides to lead you quickly through many tasks, including:

- Creating new applets
- Creating new **program elements**. In VisualAge for Java, a program element is one of the following:
  - **Project**: the top-level program element in VisualAge for Java. A project contains packages.
  - **Package**: the Java language construct. Packages contain classes and interfaces.
  - **Class**: the Java language construct. Classes contain methods and fields.
  - **Interface**: the Java language construct. Interfaces contain methods and fields. The fields in interfaces must be static final fields.
  - **Method**: the Java language construct.
- Creating features for JavaBeans
- Importing code from the file system and exporting code to the file system

Create industrial-strength Java programs

VisualAge for Java gives you the programming tools that you need to develop industrial-strength code. Specifically, you can:

Use the completely integrated visual debugger to examine and update code while it is running

Build, modify, and use JavaBeans

Browse your code at the level of project, package, class, or method

Maintain multiple editions of programs

VisualAge for Java has a sophisticated code management system that makes it easy for you to maintain multiple editions of programs. Editions of your code are organized into two areas:

The **workspace** contains the code that you are currently working on, as well as any class libraries that your code uses.

The **repository** contains all editions of all program elements.

When you want to capture the state of your code at any point, you can **version** an edition. This marks the particular edition as read-only and allows you to give it a name.

Key concepts

This section gives you the basic definitions that you need to get started.

Development with a repository

Within the VisualAge for Java environment, you do not manipulate Java code files. Instead, VisualAge for Java manages your code in a database of structured objects and shows them to you as a hierarchy of program elements:

project

package

class or interface

public, default, protected, private methods

Because you are manipulating program elements rather than files, you can concentrate on the logical organization of the code without having to worry as much about file names or directory structures.

The workspace and the repository

All activity in VisualAge for Java is organized around a single **workspace**, which contains the source code for the Java programs that you are working on. The workspace also contains all the packages, classes, and interfaces that are found in the standard Java class libraries and other class libraries that you may need.

While you work on code in the workspace, the code is automatically stored in a **repository**. In addition to storing all the code that is in the workspace, the repository contains other packages that you can add to the workspace if you need to use them.

In VisualAge for Java, you can manage the changes that you make to a program element by creating **editions** of the program element. The workspace contains at most one edition of any program element. The repository, on the other hand, contains all editions of all program elements. See Chapter 6, “Managing editions” on page 79 for more details.

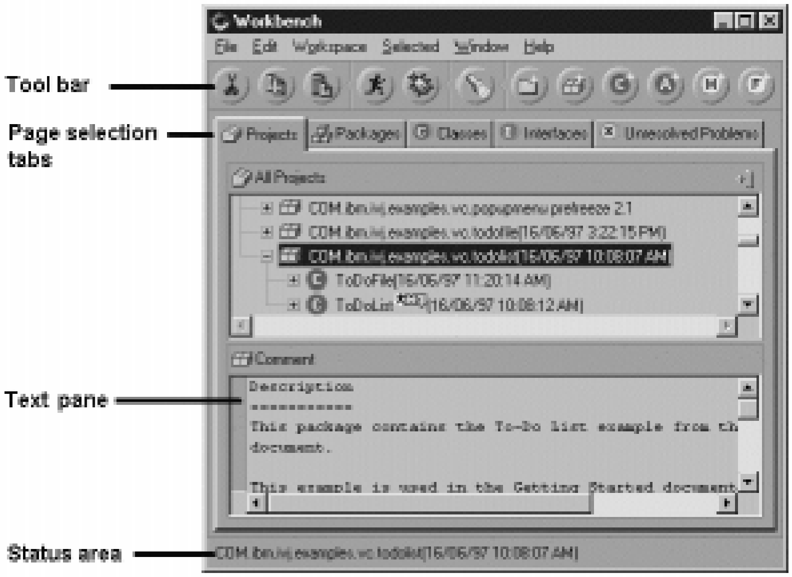
Importing and exporting code

You can easily move your code between your file system and VisualAge for Java. If you want to bring existing Java code into VisualAge for Java, you use the Import SmartGuide to specify files (or whole directory structures) that you want to bring in. VisualAge for Java compiles your code, indicates if there are any errors, and adds the appropriate program elements to the workspace.

When you want to run your program outside of VisualAge for Java, you can export it using the Export SmartGuide. VisualAge for Java creates a Java source (`*.java`) file or compiled (`*.class`) file for each class that you export.

The Workbench

VisualAge for Java gives you a variety of ways to examine and manipulate your code using different windows. The primary window you use in VisualAge for Java is called the **Workbench**. This window displays all of the program elements in the workspace.



COM.ibm.ivj.examples.vc.popupmenu.prefreeze 2.1
COM.ibm.ivj.examples.vc.todofile(16/06/97 3:22:15 PM)
COM.ibm.ivj.examples.vc.todolist(16/06/97 10:08:07 AM)
ToDoFile(16/06/97 11:20:14 AM)
ToDoList (16/06/97 10:08:12 AM)

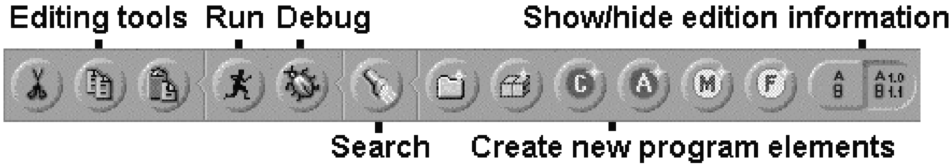
```
Description
===========
This package contains the To-Do list example from th
document.

This example is used in the Getting Started document
```

COM.ibm.ivj.examples.vc.todolist(16/06/97 10:08:07 AM)

See “Windows you can open from the Window menu” on page 91 for descriptions of the other windows in VisualAge for Java.

Tool bar

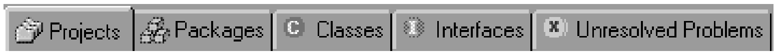
The Workbench tool bar, which is located below the menu bar, gives you easy access to the tasks you perform most frequently in the Workbench. These tasks include standard editing operations, running, debugging, searching, and manipulating program elements.



Note:

> To identify any tool in any of the tool bars in VisualAge for Java, place the mouse pointer over the tool. A label will appear that identifies the tool.

Pages in the Workbench window



Each page gives you a specific viewpoint on the code in the workspace:

> The **Projects** page displays all the projects in the workspace. You can expand projects to see the program elements inside them.
>
> The **Packages** page displays all the packages in the workspace. You can expand packages to see the program elements inside them.
>
> The **Classes** page displays all the classes in the workspace in a hierarchy rooted at `java.lang.Object`. You have the choice of displaying the hierarchy as a list or as a graphical view. You can expand a class to see what classes inherit from it.
>
> The **Interfaces** page displays all the interfaces in the workspace.
>
> The **Unresolved Problems** page displays all the classes and methods in the workspace that have unresolved problems in them. When you save code, VisualAge for Java compiles it automatically.

Note:

> It is important to make a clear distinction between the Workbench and the workspace. The *Workbench* is a window in the VisualAge for Java user interface. It displays the program elements that are in the *workspace*.

Visual programming with the Visual Composition Editor

The **Visual Composition Editor** is the portion of VisualAge for Java where you can develop programs by visually arranging and connecting software objects called **JavaBeans**, or simply **beans**. This process of creating object-oriented programs by manipulating graphical representations of components is called **visual programming**.

Beans

In VisualAge for Java, **beans** are the components that you manipulate when you program visually. These beans are Java classes that adhere to the JavaBeans specification. In the Visual Composition Editor, you select beans from a palette, specify their characteristics, and make connections between them. Beans can contain other beans and connections to beans. See “Support for JavaBeans” on page 104 for more details on the role of beans in VisualAge for Java.

There are two types of beans that you use within the Visual Composition Editor:

- A **visual bean** can be seen in your program at run time. Visual beans, such as windows, buttons, and text fields, make up the graphical user interface (GUI) of a program.
- A **nonvisual bean** does not appear in your program at run time. A nonvisual bean typically represents an object that encapsulates data and implements behavior within a program.

A bean's **public interface** determines how it can interact with other beans. The public interface of a bean consists of the following features:

- **Properties** are data that can be accessed by other beans. This data can represent any logical property of a bean, such as the balance of an account, the size of a shipment, or the label of a button.
- **Events** are signals that indicate something has happened. Opening a window or changing the value of a property, for example, will trigger an event.
- **Methods** are operations that a bean can perform. Methods can be triggered by connections from other beans.

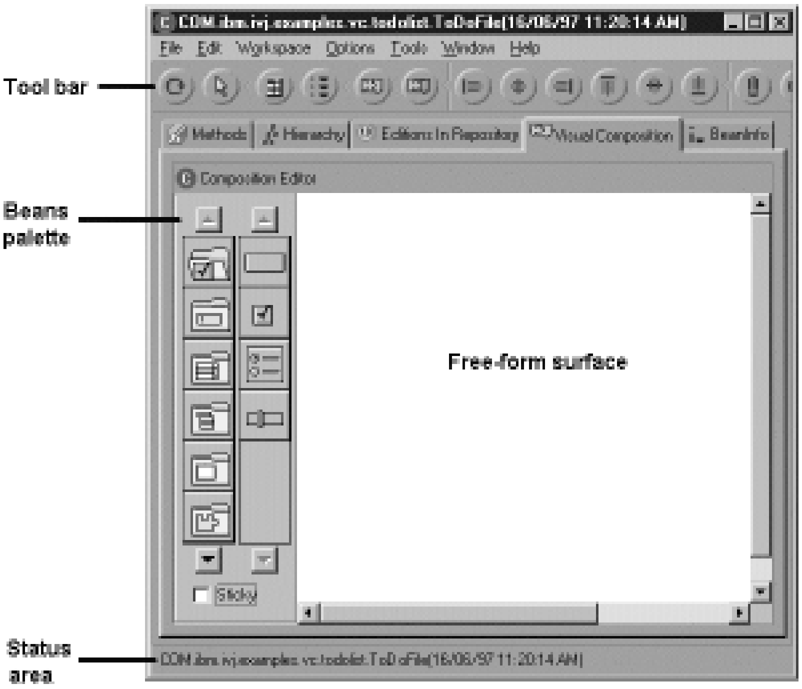
Connections

In the Visual Composition Editor, **connections** define how beans interact with each other. You can make connections between beans and between other connections. A connection has a source and a target. The point at which you start the connection is called the **source**; the point at which you end the connection is called the **target**.

The Visual Composition Editor

The Visual Composition Editor is the visual programming tool integrated with VisualAge for Java. It is one of the pages in the window that appears when you browse a class. For more details on browsing, see “Browsing” on page 97.

The Visual Composition Editor is made up of several components: the **beans palette** along the left side, the **status area** along the bottom, the **tool bar** along the top, and the **free-form surface** where you lay out the beans.

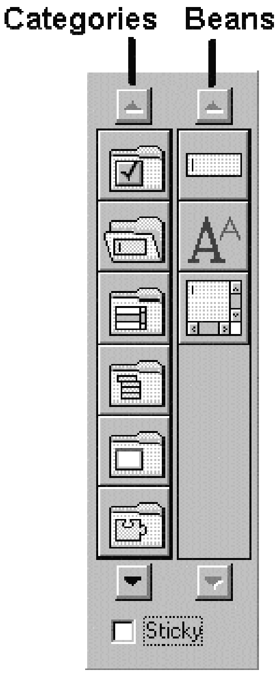

COM.ibm.ivj.examples.vc.todolist.ToDoFile(16/06/97 11:20:14 AM)
File Edit Workspace Options Tools Window Help
Tool bar
Methods
Hierarchy
Editions In Repository
Visual Composition
BeanInfo
Composition Editor
Beans palette
Free-form surface
Sticky
Status area
COM.ibm.ivj.examples.vc.todolist.ToDoFile(16/06/97 11:20:14 AM)

You use the Visual Composition Editor to construct new beans. These new beans can contain other beans as well as connections between beans. You can think of the beans you construct in the Visual Composition Editor as composite beans because they contain other beans. The composite beans you build make up your program.

Beans palette

The **beans palette**, which is located on the left side of the Visual Composition Editor, contains the set of ready-made beans that you use most frequently. The beans palette organizes the beans into **categories**.

The categories are in the left column of the beans palette, and the beans in the selected category are in the right column.



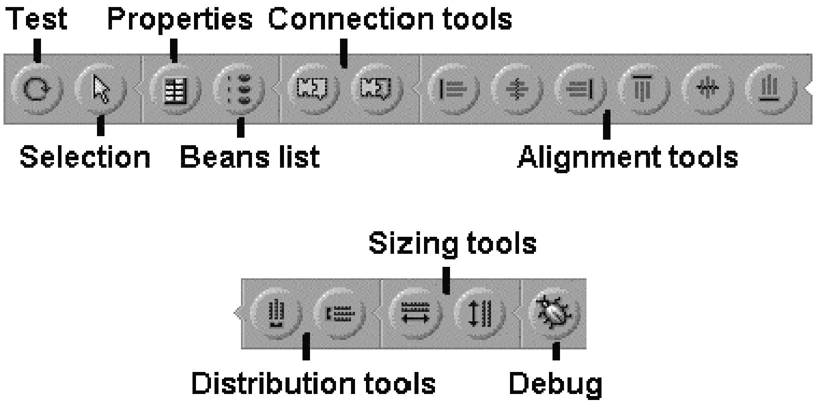
The **Status area** at the bottom of the Visual Composition Editor indicates the category and bean currently selected in the beans palette, or the bean or connection currently selected on the free-form surface.

Note:

> You can also identify a bean category or a bean by placing the mouse pointer over the icon for the bean category or bean. A label will appear that identifies the icon.

Tool bar

The **tool bar**, which is located below the menu bar of the Visual Composition Editor, provides easy access to the tools commonly used while manipulating beans. These tools help with such tasks as positioning beans, sizing beans, showing and hiding connections between beans, and testing your program.



Most of the tools in the tool bar act on the beans that are currently selected in the free-form surface. If no beans are selected for a tool to act on, the tool is unavailable.

Note:

The **Tools** menu also provides access to these tools.

Free-form surface

The large open area in the Visual Composition Editor is called the **free-form surface**. You use the free-form surface as the visual programming area where you construct your program. You cannot drop nonvisual beans on top of visual beans.

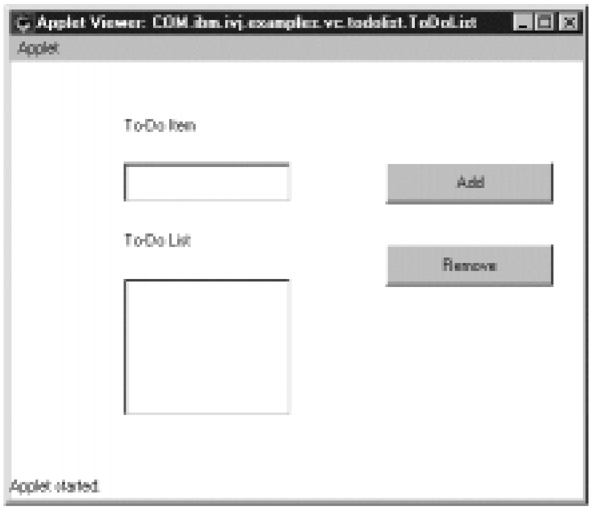
Regardless of the type of bean, every bean has a pop-up menu that contains options you can use to modify or work with that bean.

Chapter 2. Building your first applet

Introduction to building your first applet

This section leads you through building your first applet in VisualAge for Java: a To-Do List.

You will create a To-Do List applet, which consists of a bean (a **composite** bean) that is made up of many other beans. The applet has a TextField bean for entering a To-Do item and a List bean for displaying the To-Do items. There are also two Button beans for adding and removing items from the list. The user interface for the completed To-Do List applet looks like this:



In the completed applet, you type an item into the **To-Do Item** field and select **Add**. This adds the item to the **To-Do List**. If you select an item from the **To-Do List** and select **Remove**, the item is removed from the **To-Do List**.

Getting started with your first applet

If you haven't already installed VisualAge for Java, refer to the installation documentation in the `install.txt` file on the product CD for information on how to install the product. The VisualAge for Java installation program installs all the files that are necessary for your development environment.

Starting VisualAge for Java

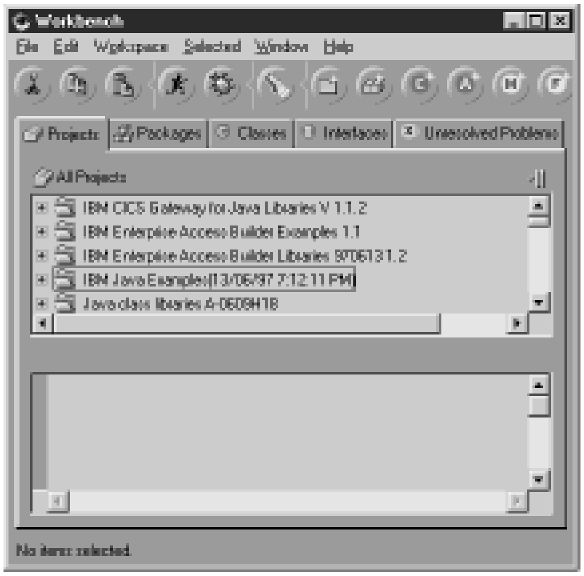
You can start VisualAge for Java by doing one of the following:

For OS/2, from the VisualAge for Java folder, double-click the **IDE** icon:

For Windows** 95 and Windows NT**, select **VisualAge for Java** then **IDE** from the **Start -> Programs** menu.

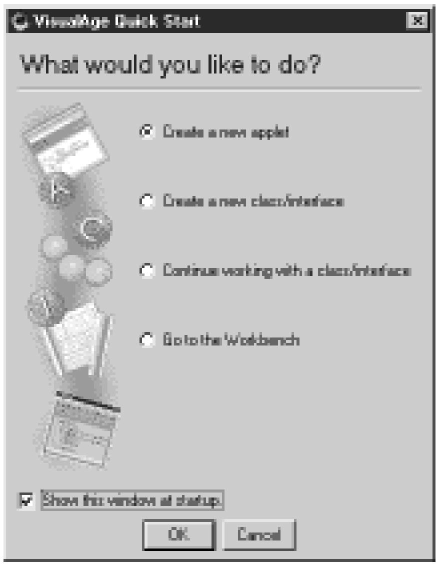
Now that you are up and running

After you start VisualAge for Java, the Workbench window appears:


Workbench
File Edit Workspace Selected Window Help
Projects
Packages
Classes
Interfaces
Unresolved Problems
All Projects
IBM CICS Gateway for Java Libraries V 1.1.2
IBM Enterprise Access Builder Examples 1.1
IBM Enterprise Access Builder Libraries 970613 1.2
IBM Java Examples(13/06/97 7:12:11 PM)
Java class libraries A-0608H18
No items selected

The Workbench window is used for accessing other windows, creating program elements, and viewing the contents of program elements.

Next, the VisualAge Quick Start window appears:


VisualAge Quick Start

What would you like to do?

● Create a new applet

○ Create a new class/interface

○ Continue working with a class/interface

○ Go to the Workbench

☑ Show this window at startup.

The VisualAge Quick Start window provides a fast path to creating applets, classes, and interfaces. To create your To-Do List applet, you'll use the VisualAge Quick Start window and the Create Applet SmartGuide.

Using a SmartGuide

For the To-Do List applet, you'll create an applet as well as a project and a package to contain your work. You'll create these using a SmartGuide that you access from the VisualAge Quick Start window.

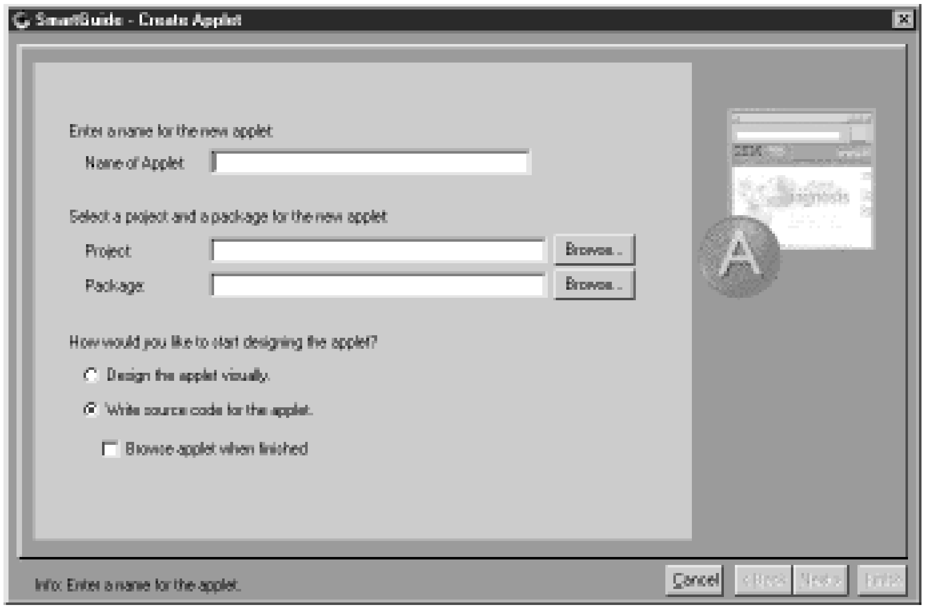
When you manipulate your new applet in the Visual Composition Editor, you will be visually manipulating JavaBeans. These JavaBeans (or, simply, **beans**) are represented as classes when you examine your applet in the Workbench.

VisualAge for Java suggests that you give your applets (and all other classes) names that begin with a capital letter. Class names are case-sensitive, and cannot contain spaces. If a class name consists of multiple words, do not type spaces between the words, but instead capitalize the first letter of each word. For example, *ToDoList*.

Creating an applet, a project, and a package

To open the SmartGuide, from the VisualAge Quick Start window:

1. Select **Create a new applet**, if it is not already selected.
2. Select **OK**. The SmartGuide - Create Applet window opens.



SmartGuide - Create Applet

Enter a name for the new applet

Name of Applet:

Select a project and a package for the new applet

Project:

Package:

How would you like to start designing the applet?

○ Design the applet visually.

● Write source code for the applet.

□ Browse applet when finished

Info: Enter a name for the applet.

In the SmartGuide - Create Applet window, follow these steps to create your applet:

1. In the **Name of Applet** field, type a name, such as *ToDoList*.
2. In the **Project** field, type a project name, such as *My ToDoList project*.
3. In the **Package** field, type a package name, such as *todolist*.
4. Select **Design the applet visually**.
5. Select **Finish**.

A secondary window opens, asking if you want to create a project and a package having the names you specified. Select **Yes**.

VisualAge for Java creates a project, a package, and an applet, then opens the Visual Composition Editor on the applet.

Using the Visual Composition Editor

When the Visual Composition Editor opens, you can begin visually constructing your To-Do List applet. The To-Do List applet consists of a bean that will contain several visual beans.

Working with beans

When working with beans in VisualAge for Java, you will use the following fundamental techniques: dragging beans, selecting multiple beans, and displaying pop-up menus.

Dragging beans

To drag a bean, click and hold down the appropriate mouse button (In OS/2, use mouse button 2 to drag a bean; in Windows, use mouse button 1). Move the mouse pointer to where you want to position the bean; then release the button.

Selecting multiple beans

To select multiple beans, hold down the control (Ctrl) key and click mouse button 1 on the items you want to select. This is referred to as a **selection set**.

Note:

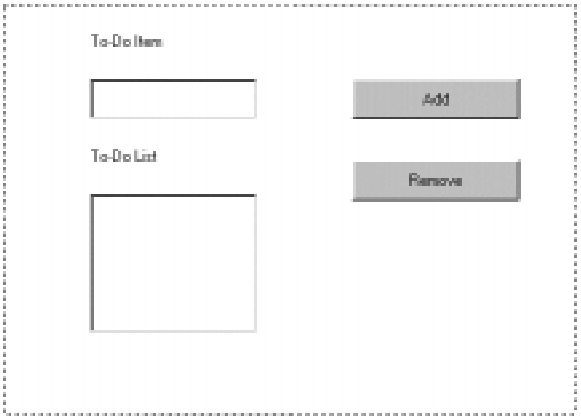
> Only beans that can be operated upon as a set can be contained in a selection set. A set of beans placed within a window, for instance, can be selected together for the purpose of sizing or alignment. However, you could not select the window and one of the beans together.

Displaying bean pop-up menus

To display a pop-up menu, click mouse button 2 on the object.

Building the To-Do List applet

When the Visual Composition Editor opens for a new visual bean, such as ToDoList, it already contains an Applet bean. The Applet bean is represented as a dashed rectangle on the free-form surface. To build the rest of the user interface, you must add several other visual beans. When you have finished creating the user interface for the To-Do List applet, the free-form surface of the Visual Composition Editor should look like this:



To make a user interface that looks like this, you need to add the remaining beans and to size and align them.

Note: As you add beans to the applet, you may find that the default Applet bean is too small to accommodate all the other beans. If this happens, you can resize the Applet bean by selecting it and dragging on one of the selection handles using mouse button 1.

Adding TextField and Label beans

To create the text field in which the To-Do items are typed, you need to add a TextField bean and a Label bean:

1. Select the **Data Entry** category: from the left column of the beans palette, and then select the **TextField** bean: from the right column of the beans palette.

   The information area at the bottom of the Visual Composition Editor displays **Category: Data Entry Bean: TextField**, reflecting the current selection in the beans palette.

2. Move the mouse pointer over the Applet bean (the dashed rectangle on the free-form surface). The pointer changes to a cross-hair, indicating that it is now loaded with the bean you selected. Click mouse button 1 where you want to add the TextField.

   If you accidentally picked the wrong bean and have not dropped the bean into the Applet yet, select the correct bean or select the **Selection** tool: from the tool bar to unload the mouse pointer.

   After you have added the TextField bean to the Applet, you can move it to a new location by dragging it with the mouse.

3. Select the **Data Entry** category, if it is not already selected, then select the **Label** bean: and add a Label just above the TextField.

   Don't worry about their exact positions. Later you'll learn how to use the tools from the tool bar to match sizes and align beans.

Changing the text of a Label

Change the text of the Label to **To-Do Item** by directly editing the text as follows:

1. Hold down the Alt key and click mouse button 1 on the Label. The text of the Label is highlighted, indicating that you can type over the text.

2. Type *To-Do Item* then click mouse button 1 in an open area of the free-form surface or press Shift+Enter.

   The Label now contains the text **To-Do Item**.

Adding a List

To create the list in which the To-Do items are displayed, you need to add a List and another Label:

1. Select the Lists category: . Then select the List bean: and add it to the Applet below the TextField.
2. You can quickly add a Label for the List by copying the one for the TextField. To copy a bean:

   Select the bean.

   From the **Edit** menu, select **Copy**.

   From the **Edit** menu, select **Paste**. The pointer changes to a cross-hair, indicating that it is now loaded with the bean you selected.

   Click mouse button 1 just above the List to add the new Label.
3. Change the new Label's text to *To-Do List* by directly editing it like you did for the TextField Label.

Tip:

You can also copy a bean using the Ctrl key. Position the mouse pointer over the Label bean, hold down the Ctrl key, and drag the copy of the bean to just above the List bean.

Adding Buttons

To add and remove items from the To-Do list, you need to add two Buttons:

1. To add more than one bean at a time, select the **Sticky** check box, located just below the beans palette.
2. Select the **Buttons** category: . Then select the **Button** bean: . Add a Button to the Applet to the right of the TextField.

   Notice that the mouse pointer remains a cross-hair, indicating that the mouse pointer is still loaded with the Button bean. Click mouse button 1 below the Button you just added to add another Button.
3. Deselect **Sticky** or select the **Selection** tool: from the tool bar to unload the mouse pointer.
4. Change the top button's text to **Add** and the bottom button's text to **Remove** by directly editing them.

Congratulations! You have just created your first user interface using VisualAge. Next, you need to size and align the beans within the To-Do List applet.

Sizing and aligning visual beans

To clean up the appearance of your user interface, use the sizing and aligning tools from the tool bar on the Visual Composition Editor. The tool bar provides several different tools for sizing and aligning beans. You'll learn a great deal about how to use them by experimenting with the different tools.

The following steps explain how to match the size of two beans, align the beans with other beans, and evenly distribute the beans within another bean. You'll learn more about sizing and changing beans in “Manipulating beans” on page 57.

Sizing, aligning, and distributing beans

The order in which you size, align, and distribute the beans is not always important. Usually, you start with the upper left corner and work your way through all the beans in the window.

To size the List so it matches the width of the TextField, do the following:

1. Select the List.
2. Hold down the Ctrl key to select multiple items and select the TextField using mouse button 1.
3. Select the **Match Width** tool: from the tool bar.

   Because the TextField was selected last, it becomes the primary selection for the match width operation. The width of the List is changed to match the width of the TextField.

Note:

The primary selection has solid selection handles. The other selected items have outlined selection handles.

To size and align the **Add** button and the **Remove** button, do the following:

1. Resize the **Remove** button to an appropriate size for the applet.
2. Select the **Add** button, hold down the Ctrl key and select the **Remove** button using mouse button 1. Then, select the **Match Width** tool: from the tool bar.
3. Because the buttons remain selected, you can now align their left edges by selecting the **Align Left** tool: from the tool bar.

To align the left side of the TextField, List, and Labels, do the following:

1. Move the Label for the TextField to the position you want it in the applet.
2. Select the TextField and List and their associated Labels, making sure to select the Label of the TextField last.

   By selecting the TextField Label last, you make it the primary selection for the alignment operation.
3. Select the **Align Left** tool: from the tool bar.

4. Because the TextField, List and Labels are still selected, you can evenly distribute them in the window by selecting the **Distribute Vertically** tool: from the tool bar.

You have now completely finished the user interface of your To-Do List applet.

Saving your work so far

Before you continue, it is a good idea to save the work you have done so far. To save the current state of your work, select **Save Bean** from the **File** menu. VisualAge for Java generates the Java code that implements the layout that you have created in the Visual Composition Editor.

Note:

> The entire applet that you are creating is a bean. When you select **Save Bean** from the **File** menu, you are saving the entire applet.

Correcting mistakes

If you make a change in the Visual Composition Editor and then decide that you should have left things as they were, select **Undo** from the **Edit** menu to restore your work to its previous state. You can undo as many operations as you want, all the way back to when you opened the Visual Composition Editor for the current bean.

If you undo an operation and then decide that you did the right thing in the first place, select **Redo** from the **Edit** menu. **Redo** will restore the view to the state it was in before the last **Undo**. As soon as you close the Visual Composition Editor for your bean, you lose any ability to undo or redo any changes made.

Connecting beans

Now that you have added the visual beans to create the user interface, the next step is connecting them.

Event-to-method and parameter-from-property connections

This is a short discussion of the event-to-method connections used in this example. It is not necessary for you to make the connections as you follow along in this text. Step-by-step instructions are provided in the next section.

The behavior of the To-Do List applet is to have the text entered in the TextField added to the list when the **Add** button is selected, and to have the selected item in the list removed when the **Remove** button is selected. To do this, you need to make **event-to-method connections** between the Buttons and the TextField and the List.

Because selecting a button signals an *actionPerformed(java.awt.event.ActionEvent)* event and adding an item to the list is performed by the *addItem(java.lang.String)* method, the event-to-method connection to add an item to the list will be between the **Add** Button's *actionPerformed(java.awt.event.ActionEvent)* event and the List's *addItem(java.lang.String)* method. Removing an item from the list is performed by connecting the **Remove** Button's *actionPerformed(java.awt.event.ActionEvent)* event to the *remove(java.lang.String)* method of the List.

Simply adding these two event-to-method connections does not actually cause anything to be added to or removed from the list because both the *addItem(java.lang.String)* and *remove(java.lang.String)* methods require a parameter that specifies what object is to be added to or removed from the list.

We'll specify the parameters for the **event-to-method** connections by creating **parameter-from-property** connections.

Because the object (the text) entered in the TextField is available through the TextField's *text* property and the parameter of the *addItem(java.lang.String)* method is specified by the *item* property of the event-to-method connection, the connection that specifies what to add to the List will be between the TextField's *text* property and the **Add** event-to-method connection's *item* property. Because the item selected in the List is available through the List's *selectedItem* property and the parameter of the *remove(java.lang.String)* method is specified by the *item* property of the event-to-method connection, the connection that specifies what to remove from the List will be between the List's *selectedItem* property and the **Remove** event-to-method connection's *item* property.

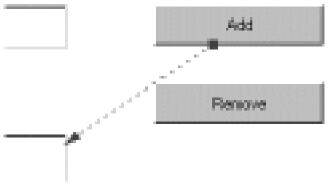
Making the connections

To make the event-to-method connection for the **Add** button, do the following:

1. Select the **Add** button, then click mouse button 2. In the pop-up menu that appears, select **Connect** then **actionPerformed(java.awt.event.ActionEvent)**.

   The mouse pointer changes, indicating that you are in the process of making a connection. If you accidentally started the wrong connection, press the Esc key to cancel.

2. To complete the connection, click mouse button 1 on the List and then select **addItem(java.lang.String)** from the pop-up menu that appears. A dashed line appears, which means that more information is necessary. In this case, the value of the parameter for the *addItem(java.lang.String)* method is missing.

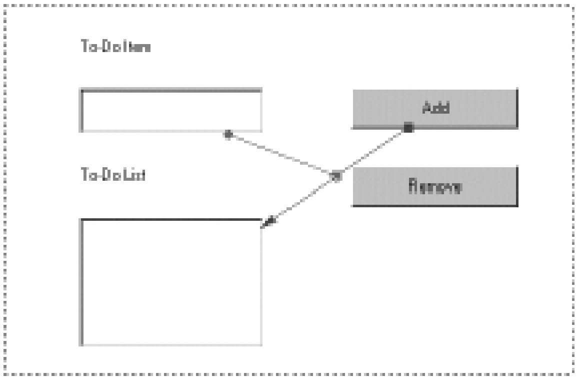
3. To make the **parameter-from-property connection** that specifies what to add to the list, follow these steps:

   Move the mouse pointer on top of the dashed event-to-method connection line.

Click mouse button 2. Select **Connect** then **item** from the pop-up menu that appears.

Click mouse button 1 on the TextField, and then select **text** from the pop-up menu that appears.

When you finish, the connections should look like this:



To make the event-to-method connection for the **Remove** button, do the following:

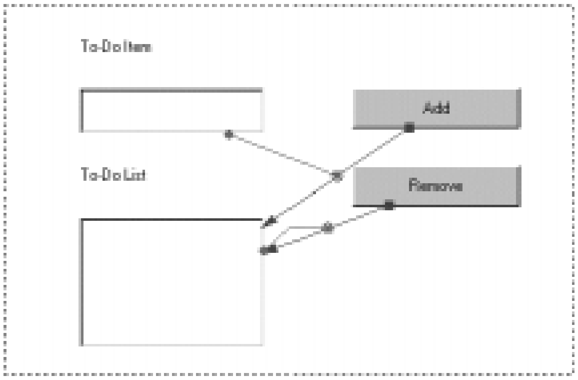
1. Select the **Remove** button, then click mouse button 2. In the pop-up menu that appears, select **Connect** then **actionPerformed(java.awt.event.ActionEvent)**.
2. Click mouse button 1 on the List, then select **remove(java.lang.String)** from the pop-up menu that appears. Again, a dashed line appears, which means that more information is necessary. In this case, the value of the parameter for the *remove(java.lang.String)* method is missing.
3. To make the **parameter-from-property connection** that specifies what to remove from the list, follow these steps:

   Move the mouse pointer on top of the dashed connection line.

   Click mouse button 2, then select **Connect** then **item** from the pop-up menu that appears.

   Click mouse button 1 on the List, then select **selectedItem** from the pop-up menu that appears.

Your connections should now look like this:



With the user interface complete and the behavior of the applet defined by the connections between the visual beans, you are now ready to save and test your work.

Saving and testing the To-do List applet

Before testing your ToDoList applet, you should save the work you have done.

When you save changes to a bean, you are replacing the old specification of the bean with a new one. When you do this, VisualAge for Java will be using the new specification of the bean for all new uses of it. You should save your changes to a bean periodically as you are working with it and when you have finished editing it.

Saving your visual bean

In the Visual Composition Editor, do the following:

To save your bean, from the **File** menu select **Save bean**.

A message box appears saying that your bean is being saved and that *runtime code* is being generated. This generated runtime code is what VisualAge uses to create your bean when you run your application.

Testing the applet

Now that your work is saved, you can test your To-Do List applet.

1. To begin testing your applet from within the Visual Composition Editor, select **Test** from the tool bar: When the Settings window for the applet opens, simply select **Run** to display the applet.
2. When the To-Do List applet appears, experiment with it to ensure that it behaves the way you expect it to. For the To-Do List applet, you need to ensure you can add typed items to the list and remove selected items from the list.

Be sure to close the applet window when you have completed your testing.

At any time, you may return to the Visual Composition Editor and make changes, save the changes, then test the applet again.

Congratulations! Your To-Do List applet is finished.

Saving your workspace

Before you continue, save your workspace. When you save your workspace, you are saving the current state of all the code that you are working on and the state of any windows that you currently have open. To save your workspace:

Select **Save Workspace** from the **File** menu.

Chapter 3. Adding state checking to your applet

Introduction to adding state checking to your applet

There is one piece of unfinished business left over from the To-Do List applet that you created. To keep the applet as simple as possible, we did not include any kind of **state checking**. The **Add** and **Remove** buttons are always available. This is not the ideal behavior for the applet. For example, a user should only be able to select the **Remove** button if there is an item selected in the **To-Do List**.

This section leads you through the steps to add state checking to your applet. It's a chance for you to review what you learned when you created the To-Do List applet and to learn a bit more about how the Visual Composition Editor works. This section only deals with the **Remove** button, but if you want to experiment, you can try to add the same kind of state checking for the **Add** button.

Before we update the applet to include state checking, we'll go through the steps to find your applet and to create a versioned edition of it.

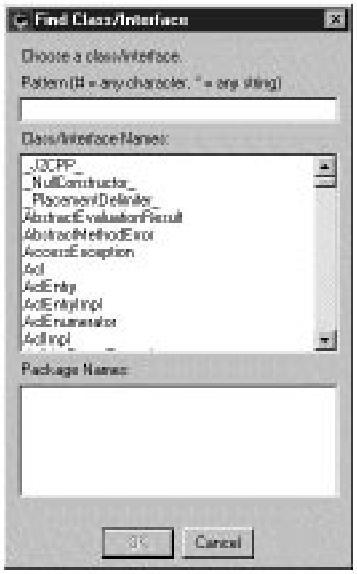
Note:

> This section assumes that you have completed the steps described in Chapter 2, “Building your first applet” on page 9. You should now have a completed, working To-Do List applet. If you have not done so already, please complete the steps to create the basic To-Do List applet.

Finding your To-Do List applet in the Workbench

Before you can add state checking to your To-Do List applet, you may need to find it. When you created it in the Visual Composition Editor, you may not have kept track of where VisualAge for Java was putting the code it generated to implement the applet. Don't worry. VisualAge for Java gives you powerful search capabilities for finding program elements. These capabilities are described in more detail in “Searching” on page 94. For now, here is quick way to find your To-Do List applet:

1. In the Workbench window, select the **Projects** page.
2. From the **Selected** menu, select **Go To** then **Class/Interface**. The Find Class/Interface secondary window appears:



3. Enter the name of your To-Do List applet (for example, *ToDoList*) in the **Pattern** field. As you enter the name, the **Class/Interface Names** list changes to include only the classes and interfaces that match what you have entered so far.
4. Select the name of your To-Do List applet from the **Class/Interface Names** list. If packages are listed in **Package Names**, it means that more than one package has a class with the name you specified. Select the package in which you created the To-Do List applet and select **OK**.
5. The list of projects is updated. The project and package that contain your applet are expanded, and the class for your applet is selected.

Now you have found the class for your applet, you are ready to version it.

Versioning an edition of your applet

When you **version** an edition of a program element, you give it a name and explicitly save its current state. When you make more changes to the code and save these changes, a new edition is created based on this versioned edition. If you decide you want to undo these changes or try a different set of changes, you can simply return to the versioned edition. For more details on editions and versioned editions, see Chapter 6, “Managing editions” on page 79.

Before you make any changes to your To-Do List applet class, version it so you can return to it if you need to do so. To version the class for your applet:

1. Ensure the class for your applet is selected, then select **Version** from the **Selected** menu. The Versioning Selected Items SmartGuide appears.

2. Ensure that **Automatic** is selected and select **Finish**. If **Show Edition Names**: is selected in the tool bar, a version name appears next to the class.

The next time you modify this class and save it, VisualAge for Java creates a new edition based on the code in this versioned edition. If you run into problems while you are making updates to your applet, you can return to your working To-Do List applet by returning to the versioned edition you just created.

Adding state checking to your applet

Now that you have versioned an edition of your applet, you are ready to add state checking.

Desired behavior of the Remove button

Currently, the **Remove** button is always enabled, even if nothing is selected to remove. Here is how the **Remove** button should work:

When the applet starts, the **Remove** button should be disabled.

When there is an item selected in the **To-Do List**, the **Remove** button should be enabled.

When no items are selected in the **To-Do List**, the **Remove** button should be disabled.

Overview of adding the desired behavior to the Remove button

To get the desired behavior for the **Remove** button, you need to:

Open the To-Do List applet in the Visual Composition Editor.

Set the properties of the **Remove** button so it is disabled when the applet first starts.

Create a new method that checks if an item is selected in the **To-Do List**.

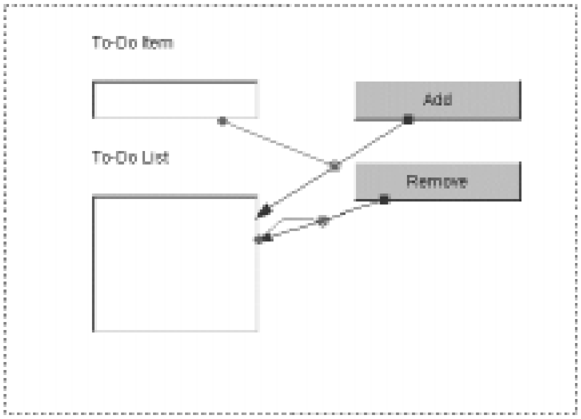
Add a connection between the **To-Do List** and the **Remove** button to enable the button when an item is selected in the **To-Do List**.

Add a connection between the **Remove** button and itself to disable it when there are no items selected in the **To-Do List**.

Open your To-Do List applet in the Visual Composition Editor

First, open your To-Do List applet class in the Visual Composition Editor:

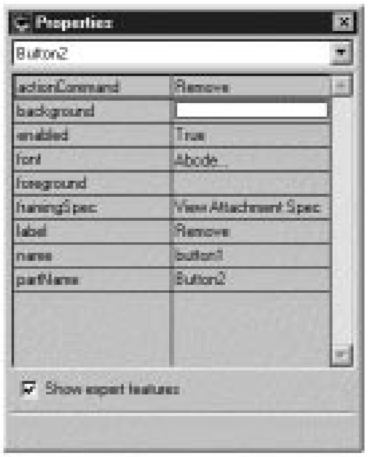
1. Select the class for your To-Do List applet in the Workbench.
2. Select **Open** from the **Selected** menu. A browser opens for the To-Do List applet class.
3. Select the **Visual Composition** page. The free-form surface should look like this:



Set the properties of the Remove button

Now, set the properties of the **Remove** button so it is disabled when the applet starts:

1. Select the **Remove** button and click mouse button 2. Select **Properties** from the pop-up menu that appears. The Properties secondary window appears.

   Button2

actionCommand	Remove
background	
enabled	True
font	Abcde...
foreground	
framingSpec	View Attachment Spec
label	Remove
name	button1
partName	Button2

   ☑ Show expert features

2. In the Properties secondary window, ensure that **Show expert features** is selected.

3. Select the field to the right of **enabled**. Select **False** from the drop down list in this field and close the Properties secondary window. The **Remove** button should now appear disabled:

Create a new method to check if an item is selected

Next, create a new method in your To-Do List applet that checks to see if any items are selected in the To-Do List. To create a new method:

1. Select the **Methods** page.
2. Select **Create Method or Constructor** from the tool bar: . The Create Method SmartGuide appears.
3. In the Create Method SmartGuide, enter the following in the Method Name field:

   ```
   boolean enableRemove(List checkList)
   ```
4. This specifies a method that takes one parameter (a List) and returns a boolean value.
5. Select **Finish** to generate the method.
6. Select the new *enableRemove(java.awt.List)* method from the **Methods** list and add the code to implement it. If you are viewing this document online, you can select the following code, copy it, and paste it into the **Source** pane. The finished method should look like this:

   ```
   public boolean enableRemove(java.awt.List checkList) {
    if (checkList.getSelectedIndex() < 0)
         return false;
    else
         return true;
   }
   ```
7. To save this new method, click mouse button 2 in the **Source** pane and select **Save** from the pop-up menu that appears.

This simple method calls the *getSelectedIndex* method for its checkList parameter. If *getSelectedIndex* returns -1, there are no items selected in the list and *enableRemove(java.awt.List)* returns false. Otherwise, *enableRemove(java.awt.List)* returns true.

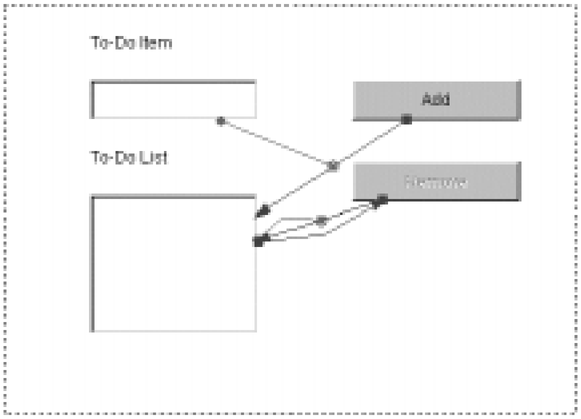
Add a connection to enable the Remove button

Next, add the connection that enables the **Remove** button when an item is selected in the **To-Do List**:

1. Select the **Visual Composition** page.
2. Select the List and click mouse button 2. Select **Connect** then **itemStateChanged(java.awt.event.ItemEvent)** from the pop-up menu that appears. The mouse pointer changes to indicate that you are in the process of making a connection.
3. Complete the connection by clicking mouse button 1 on the **Remove** button. Select **enabled** from the pop-up menu that appears. A dashed connection appears between the list and the **Remove** button.

4. The connection is incomplete because you need to provide a value for the *enabled* property of the button. To complete the connection, select it and click mouse button 2. Select **Properties** in the pop-up menu that appears. The Event-to-method connection Properties window appears.
5. In the Properties window, select **Set Parameters**. The Constant Parameter Settings window appears. Select the **value** field, and select **true** in the drop-down menu that appears, and select **OK**.
6. Select **OK** in the Properties window.

The free-form surface should look like this:



Now, every time an item is selected in the list, the **enabled** property of the **Remove** button is set to true.

Add connections to disable the Remove button

Finally add the connections that disable the **Remove** button when no items are selected in the **To-Do List**. These connections will set the *enabled* property of the Remove button to the return value of the *enableRemove(java.awt.List)* method (with the List as a parameter for *enableRemove*).

Follow these steps to make the connections:

1. Select the **Remove** button, then click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears.
2. The mouse pointer changes. Click mouse button 1 on the **Remove** button. Yes, this is a connection between the Remove button and itself. Select **enabled** from the pop-up menu that appears. An incomplete connection (we'll call it **connection A**) appears between the button and itself. To complete the connection, you need to specify a value for the **enabled** property.
3. Select the connection you just made (**connection A**) and click mouse button 2. Select **Connect** then **value** from the pop-up menu that appears. Now move the

mouse pointer onto the free-form surface outside of the applet outline and click mouse button 1. Select **Parameter from Script** in the pop-up menu that appears. The Connect parameter named secondary window appears. It lists all the methods in the applet class, including *enableRemove(java.awt.List)*, the method you just created.

4. Select **boolean enableRemove(java.awt.List)** in the list and select **OK**. Now **connection A** is complete. An incomplete connection appears between **connection A** and the edge of the free-form surface. We'll call this **connection B**. **Connection B** specifies that the value of **enabled** in **connection A** is the return value of *enableRemove(java.awt.List)*. To complete **connection B**, you need to supply a List parameter for the *enableRemove(java.awt.List)* method. To supply this parameter:

   Select the connection and click mouse button 2. Select **Connect** then **checkList** from the pop-up menu that appears. Notice that *checkList* is the name you gave to the parameter when you created the *enableRemove(java.awt.List)* method.

   Move the mouse pointer to the List and click mouse button 1. Select **this** from the pop-up menu that appears. Now you have specified that the list is passed as the parameter to *enableRemove(java.awt.List)*. We'll call this **connection C**.

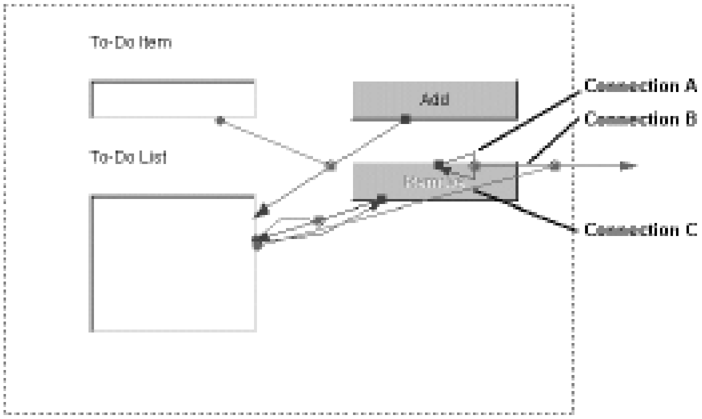
You have completed all the connections to disable the Remove button when nothing is selected in the **To-Do List**. Before we continue, let's review this last set of connections:

Connection A sets the enabled property of the Remove button.

Connection B provides the value of the enabled property for **connection A**. This value is the return value of the *enableRemove(java.awt.List)* method.

Connection C provides the parameter for the *enableRemove(java.awt.List)* method.

Now the free-form surface should look like this. The labels for the connections don't appear in the Visual Composition Editor. We've added them here to make it easier for you to identify them.



Saving and testing your changes

Before you continue, save your work and test it:

1. To save the current state of your work in the Visual Composition Editor, select **Save bean** from the **File** menu.
2. To test the changes you made, select **Test** from the tool bar:
3. When the Settings window appears, select **Run**.
4. When the applet appears, experiment with it to ensure that the behavior of the **Remove** button is correct. Ensure that the **Remove** button is disabled when the applet starts and then becomes enabled as soon as an item is selected in the **To-Do List**. Ensure that the **Remove** button becomes disabled again when all the items have been removed from the **To-Do List**.

Congratulations! You have successfully added state checking to your To-Do List applet.

Now that you have a new level of your code working, create another versioned edition of it by following the steps in "Versioning an edition of your applet" on page 24.

Chapter 4. Creating the To-Do File program

Introduction to creating the To-Do File program

In the previous section, you added state checking to your simple To-Do List applet. This section leads you through the steps of modifying your simple To-Do List applet so that it can save the To-Do lists to named files and open files containing To-Do lists. This revised program is called the *To-Do File program*.

As you modify your applet, you will learn about:

- Creating new classes
- Creating new methods
- Adding business logic code in the Visual Composition Editor
- Updating the user interface
- Running code as an applet or an application

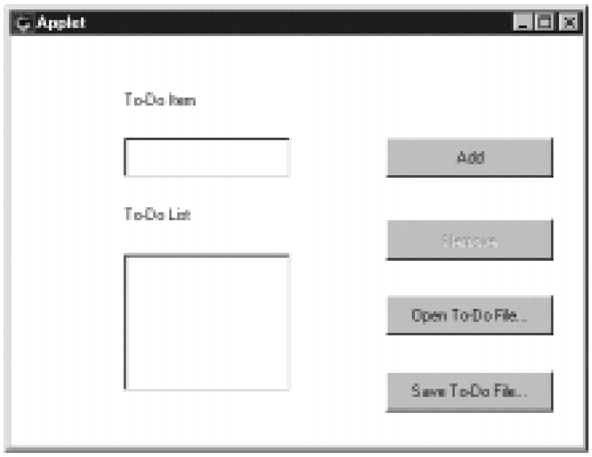
Note:

> This section assumes that you have completed the steps described in Chapter 3, “Adding state checking to your applet” on page 23. You should now have a completed, working To-Do List applet with simple state checking. If you have not done so already, please complete the steps to add state checking to your To-Do List applet.

Behavior of the To-Do File program

Before jumping into the modifications that you will be making to your applet to create the To-Do File program, let's review how the finished program will work.

Here is what the To-Do File program will look like:



Like your existing applet, the To-Do File program adds the text in the **To-Do Item** field to the **To-Do List** when you select the **Add** button. When you select the **Remove** button, the program removes the selected item from the **To-Do List**.

What about the new buttons? Here is an overview of their behavior:

When you select **Open To-Do List File**, a standard file dialog is shown, from which you can select the file you want to open. If you select a file, its contents are put into the **To-Do List**.

When you select **Save To-Do List File**, a standard file dialog is also shown. In this dialog you can specify the file where you want to save your list. If you select a file for saving, the contents of the **To-Do List** are copied into this file.

In addition to the differences in interface and behavior, there is one other important difference between the To-Do List applet and the To-Do File program. Because it needs access to the file system to read and write files, the To-Do File program must be run as an application rather than an applet. Java applets are not allowed to access the file system.

Steps for creating the To-Do File program

Here is a summary of the steps that you will follow to create the To-Do File program:

1. Create a new class called ToDoFile with logic for reading and writing files.
2. Add the **Open To-Do File** and **Save To-Do File** buttons to the user interface.
3. Add the ToDoFile class to the Visual Composition Editor free-form surface.
4. Add file dialog beans.
5. Add connections from the **Open To-Do List File** button.
6. Test the program to verify your work so far.

7. Add connections from the **Save To-Do File** button.
8. Test the completed program.

The following sections describe these steps in detail.

Creating a new class

The next step in building the To-Do File program is creating the ToDoFile class. When it is complete, this class will contain the logic for the To-Do File program to:

- Read To-Do files
- Write To-Do files

Here are the individual tasks that need to be completed to create the new class:

1. Use the Create Class or Interface SmartGuide to create a skeleton for the ToDoFile class.
2. Add a method for reading To-Do files.
3. Add a method for writing To-Do files.

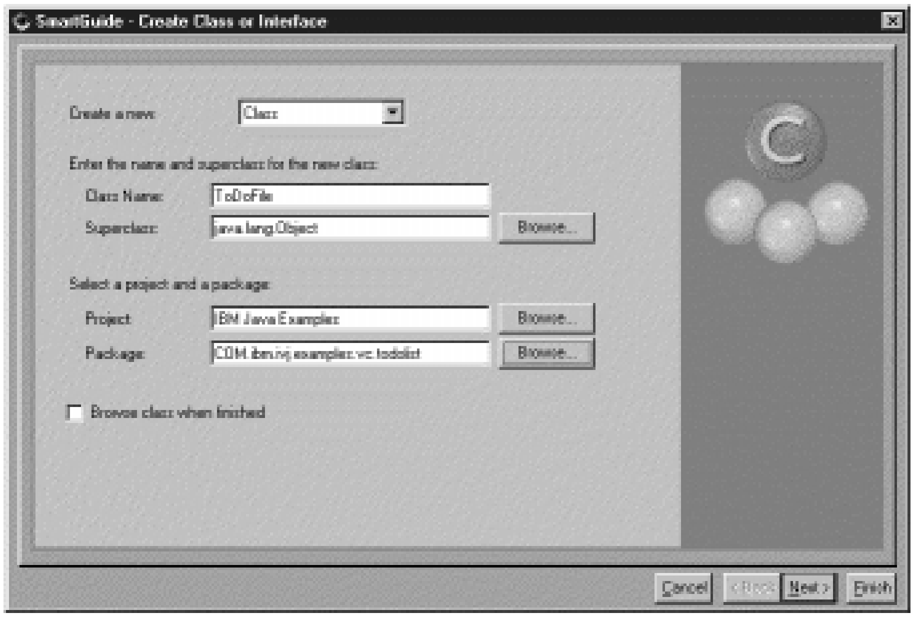
The following sections describe these tasks in more detail.

Creating a new class: creating a skeleton

Now you are ready to create the ToDoFile class in the same package as your To-Do List applet. In later steps, you will create methods in this class to read and write files. For now, you will use the Create Class or Interface SmartGuide to generate a skeleton of the class.

To generate a skeleton ToDoFile class in the same package as your To-Do List applet class:

1. Select the package that contains your To-Do List applet class, then select **New Class/Interface** from the **Selected** menu. The Create Class or Interface SmartGuide appears.



SmartGuide - Create Class or Interface

Create a new: Class

Enter the name and superclass for the new class:

Class Name: ToDoFile

Superclass: java.lang.Object

Select a project and a package:

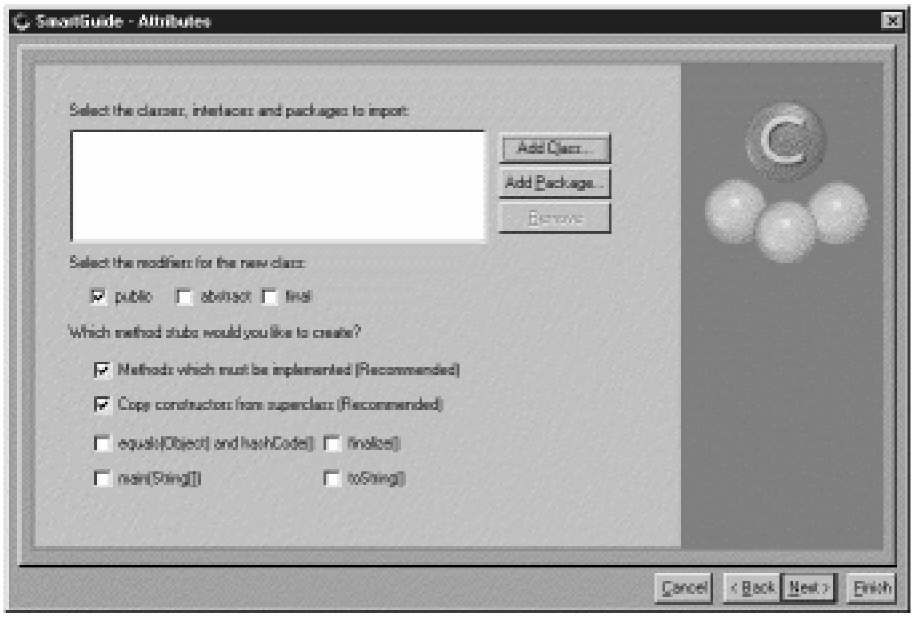
Project: IBM Java Examples

Package: COM.ibm.ivj.examples.vc.todolist

☐ Browse class when finished

You could also select **Create Class or Interface** from the tool bar: to get to the Create Class or Interface SmartGuide.

2. Ensure **Class** is selected in the **Create a new** field. Enter *ToDoFile* in the **Class Name** field and select **Next.** The Attributes page of the SmartGuide appears:



SmartGuide - Attributes

Select the classes, interfaces and packages to import:

Select the modifiers for the new class:

☑ public ☐ abstract ☐ final

Which method stubs would you like to create?

☑ Methods which must be implemented (Recommended)

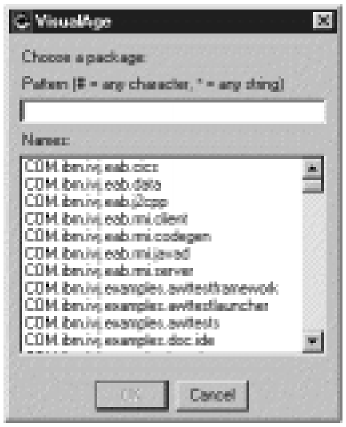
☑ Copy constructors from superclass (Recommended)

☐ equals(Object) and hashCode() ☐ finalize()

☐ main(String[]) ☐ toString()

3. The methods that you will add to the ToDoFile class require two packages to be imported: `java.awt` and `java.io`. To specify that ToDoFile imports these classes:

   Select **Add Package**. The following secondary window appears:

   
   VisualAge
Choose a package:
Pattern (# = any character, * = any string)
Names:
COM.ibm.ivj.eab.cicx
COM.ibm.ivj.eab.data
COM.ibm.ivj.eab.j2cpp
COM.ibm.ivj.eab.rmi.client
COM.ibm.ivj.eab.rmi.codegen
COM.ibm.ivj.eab.rmi.javad
COM.ibm.ivj.eab.rmi.server
COM.ibm.ivj.examples.awttestframework
COM.ibm.ivj.examples.awttestlauncher
COM.ibm.ivj.examples.awttests
COM.ibm.ivj.examples.doc.ide
OK
Cancel

   Enter *java.awt* and select **OK.**

   Select **Add Package** again. In the secondary window, enter *java.io* and select **OK**.

4. The list of imports should look like this:

   ```
   import java.awt.*;
   import java.io.*;
   ```

5. Select **Finish**.

VisualAge for Java generates the skeleton class and creates a new ToDoFile class that appears in the same package as your To-Do List applet class. The source for the new class appears in the **Source** pane.

Creating a new class: adding a method for reading files

Now that you have created a skeleton class, you are ready to begin filling it in. Let's start by creating a method called *readToDoFile* that reads an input file.

Before creating this method, let's review what it is supposed to do:

1. Accept as arguments a directory, a file name, and a List object.
2. Read the contents of the file line-by-line and add each line as an item in the List object.

Here are the detailed steps for creating this method:

1. Select the ToDoFile class.
2. Select **New Method** from the **Selected** menu. When the Method Properties SmartGuide appears, enter the following in the **Method Name** field:

```
void readToDoFile(String dirName,
 String fileName, List fillList)
```

3. This specifies a method that takes three arguments.

   `dirName` - the name of the directory that holds the file to be read

   `fileName` - the name of the file to be read

   `fillList` - the List object in the interface that receives the contents of the file

4. Select **Finish** to generate the method.

5. Select the new *readToDoFile* method and add the code to implement it. If you are viewing this document online in a browser, you can select the following code, copy it, and paste it into the **Source** pane. The finished method should look like this:

```
public void readToDoFile  (String dirName,
      String fileName, List fillList) {
 FileInputStream fileInStream = null;
 DataInputStream dataInStream;
 String result;
 // if valid directory and filenames have been passed in,
 // read the file and fill the list
 if ((dirName != null) && (fileName != null)) {
      try {
           fileInStream= new FileInputStream(dirName+fileName);
      }
      catch (IOException e) {
           System.err.println("IO exception opening To-Do File "
                +dirName+fileName);
           return;
      }
      dataInStream = new DataInputStream(fileInStream);
      // clear the existing entries from the list
      fillList.removeAll();
      try {
           // for each line in the file create an item in the list
                while ((result = dataInStream.readLine()) != null){
                if (result.length() != 0)
                     fillList.addItem(result);
                }
      }
      catch (IOException e) {System.err.println(
           "IO exception reading To-Do File "
           +dirName+fileName);}
      try {
           fileInStream.close();
           dataInStream.close();
      }
      catch (IOException e) { System.err.println(
           "IO exception closing To-Do File "
           +dirName+fileName);}
 }
      else {
           System.err.println(
      "Null file name and/or directory reading To-Do File");
      }
      return;
}
```

6. Select **Save** from the **Edit** menu to save your changes and recompile.

Before continuing with the next task, let's review the code in this method:

1. At the beginning of the method there are declarations of the fields that are used to manipulate the file and its contents, and an `if` statement that ensures neither the directory nor the file name is null:

```
FileInputStream fileInStream = null;
DataInputStream dataInStream;
String result;
// if valid directory and filenames have been passed in,
// read the file and fill the list
if ((dirName != null) && (fileName != null)) {
```

2. Next, there are statements to associate the file with a FileInputStream and to associate the FileInputStream with a DataInputStream. Using a DataInputStream makes it possible to read the file a line at a time.

```
try {
 fileInStream= new FileInputStream(dirName+fileName);
}
catch (IOException e) {
 System.err.println(
      "IO exception opening To-Do File "
      +dirName+fileName);
 return;
}
dataInStream = new DataInputStream(fileInStream);
```

3. Next, we clear the `fillList`. Then there is a loop that reads the file a line at a time into the String `result`. Then, if result is not a zero-length String, it adds `result` as an item to `fillList`:

```
fillList.removeAll();
try {
 // for each line in the file create an item in the list
 while (((result = dataInStream.readLine()) != null) ){
      if (result.length() != 0)
           fillList.addItem(result);
 }
}
catch (IOException e) {System.err.println(
 "IO exception reading To-Do File "
 +dirName+fileName);}
```

4. Finally, there are statements to close the streams associated with the file:

```
try {
 fileInStream.close();
 dataInStream.close();
}
catch (IOException e) { System.err.println(
 "IO exception closing To-Do File "
 +dirName+fileName);}
```

Creating a new class: adding a method for writing files

You have one more method to add to the ToDoFile class. This method, called *writeToDoFile*, writes an output file. Let's review what this method is supposed to do:

1. Accept as arguments a directory, a file name, and a List object

2. Write each item in the List object as a line in the file

Here are the detailed steps for creating this method:

1. Select the ToDoFile class.
2. Select **New Method** from the **Selected** menu. When the Method Properties SmartGuide appears, enter the following in the Method Name field:

   ```
   void writeToDoFile(String dirName,
    String fileName, List fillList)
   ```

3. This specifies a method that takes 3 arguments.

   `dirName` - the name of the directory that holds the file to be written

   `fileName` - the name of the file to be written

   `fillList` - the List object in the interface that contains the items that are written to the file

4. Select **Finish** to generate the method.
5. Select the new *writeToDoFile* method and add the code to implement it. If you are viewing this document online in a browser, you can select the following code, copy it, and paste it into the Source pane. The finished method should look like this:

```
public void writeToDoFile(String dirName, String fileName, List fillList) {
 FileOutputStream fileOutStream = null;
 DataOutputStream dataOutStream;
 // carriage return and line feed constant
 String crlf =
      System.getProperties().getProperty("line.separator");
 // if valid directory and filenames have been
 // passed in, write the file from the list
 if ((dirName != null) && (fileName != null)) {
      try {
           fileOutStream =
                new FileOutputStream(dirName+fileName);
      }
      catch (IOException e) {
           System.err.println(
                "IO exception opening To-Do File "
                +dirName+fileName);
           return;
      }
      dataOutStream = new DataOutputStream(fileOutStream);
      // for every item in the list,
      // write a line to the output file
      for (int i = 0; i < fillList.countItems(); i++) {
           try {
                dataOutStream.writeBytes(fillList.getItem(i)+crlf);
           }
           catch (IOException e) { System.err.println(
                "IO exception writing To-Do File "
                +dirName+fileName);}
      }
      try {
           fileOutStream.close();
           dataOutStream.close();
      }
      catch (IOException e) { System.err.println(
           "IO exception closing To-Do File "
           +dirName+fileName);}
 }
 else {
           System.err.println(
           "Null file name and/or directory writing To-Do File");
      }
 return;
}
```

6. Select **Save** from the **Edit** menu to save your changes and recompile.

This code is similar to the code for readToDoFile. Before continuing with the next step, let's review the loop that actually writes lines to the file:

```
for (int i = 0; i < fillList.countItems(); i++) {
 try {
      dataOutStream.writeBytes(fillList.getItem(i)+crlf);
 }
 catch (IOException e) { System.err.println(
      "IO exception writing To-Do File "
      +dirName+fileName);}
}
```

This loop goes through each item in `fillList`. Each item is appended with `crlf` (a String consisting of the line separator characters) and written to the file. The line separator characters force each item to be written on a separate line in the file.

Using the Scrapbook to test code

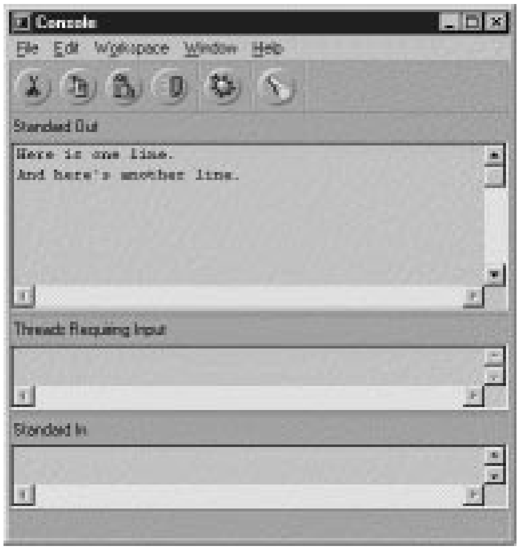
Before continuing, let's pause and consider the line separator for a moment. Suppose that you have never seen this before and you want to see how it works. You can use the **Scrapbook** window to test out a code fragment that exercises this part of your class.

To test the line separator code:

1. Select **Scrapbook** from the **Window** menu. The Scrapbook window appears.
2. Enter the following code into a page in the Scrapbook window:

   ```
   String crlf =
    System.getProperties().getProperty("line.separator");
   System.out.println(
    "Here is one line."+crlf+
    "And here's another line.");
   ```

3. Select both of these lines of code and select **Run** from the Scrapbook window tool bar: 🏃
4. Select **Console** from the **Window** menu. The Console window should look like this:

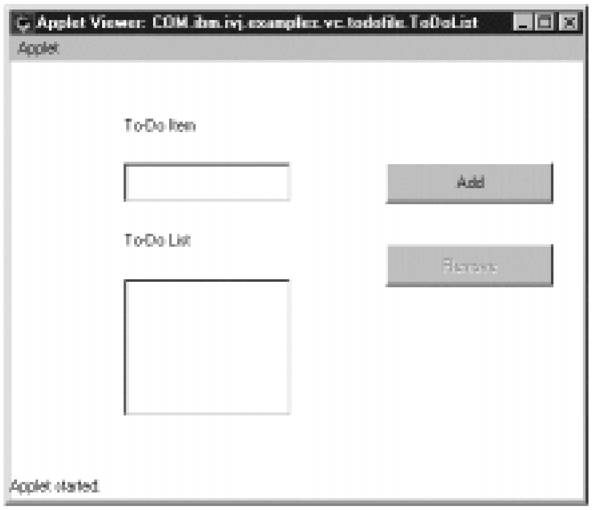


```
Here is one line.
And here's another line.
```

Notice that the line separator splits the output so that it appears on separate lines. This simple example demonstrates how you can use the Scrapbook window to try out a piece of code quickly and conveniently.

Adding buttons to the To-Do List applet user interface

You have completed all the steps to create the ToDoFile class. Now you are ready to make modifications to the user interface of the To-Do List applet. Your current To-Do List applet should look like this:



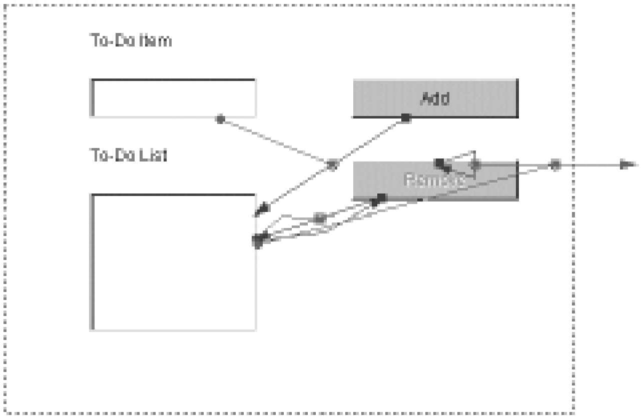
You need to add two new buttons to this user interface:

An **Open To-Do File** button to trigger opening a file to read into the **To-Do List** list

A **Save To-Do File** button to trigger saving the contents of the **To-Do List** list to a file

To add these two buttons:

1. Select the class for your To-Do List applet.
2. Select **Open** from the **Selected** menu. A browser opens for the To-Do List applet class.
3. Select the **Visual Composition** page. The free-form surface should look like this:

4. Select the **Buttons** category in the left column of the beans palette, and then select the **Button** bean in the right column. Add a Button under the existing **Remove** button.
5. Select the button you just added and change its text to **Open To-Do File**. To change the text:

   Hold down the Alt key and click mouse button 1. The text of the button is highlighted, indicating you can type over the text.

   Type *Open To-Do File...* then click mouse button 1 in an open area of the free-form surface.

6. Follow the same procedure to add another button below the one you just added. Change the text of this button to *Save To-Do File...*
7. Size the new buttons to match the width of the existing buttons:

   Select the **Save To-Do File** button. Hold down the Ctrl key to select multiple items and select the **Open To-Do File**, **Remove**, and **Add** buttons so that all four buttons are selected. The **Add** button should have solid selection handles, indicating that it is the primary selection.

   Select **Match Width** from the tool bar:

8. Align the two new buttons with the existing **Add** and **Remove** buttons:

Select the **Save To-Do File** button. Hold down the Ctrl key to select multiple items and select the **Open To-Do File**, **Remove**, and **Add** buttons so that all four buttons are selected. The **Add** button should have solid selection handles, indicating that it is the primary selection.

Select **Align Left** from the tool bar:

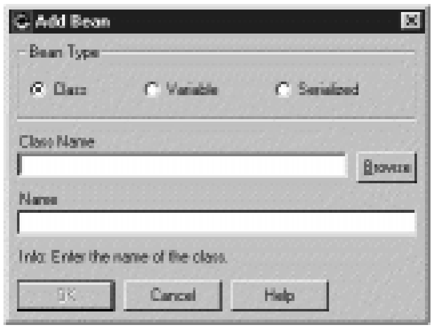
You have added the two new buttons for the To-Do File program. Now you are ready to add the ToDoFile class to the Visual Composition Editor free-form surface.

Adding the ToDoFile class to the free-form surface

Now that you have added the new buttons to the interface of the program, you need to add the other beans that these buttons are going to interact with. First, you need to add the ToDoFile class to the Visual Composition Editor free-form surface so you can create connections between the class and these new buttons. The ToDoFile bean that you create is a **nonvisual** bean because it does not appear in the user interface of the program.

To add the ToDoFile class as a bean on the Visual Composition Editor free-form surface:

1. In the Visual Composition Editor, select **Add Bean** from the **Options** menu. The Add Bean secondary window appears.



Add Bean

Bean Type

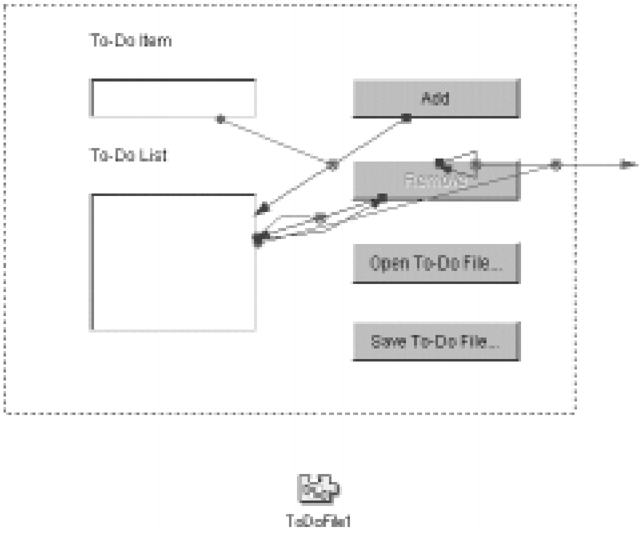
● Class ○ Variable ○ Serialized

Class Name

Name

Info: Enter the name of the class.

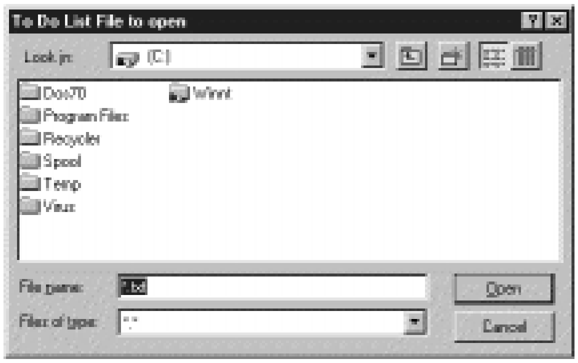
2. Ensure the Bean Type **Class** is selected and select **Browse**. The Choose a valid class secondary window appears.
3. Enter *ToDoFile* in the **Pattern** field. Select the package in which you created the ToDoFile class from the **Package Names** list and select **OK**.
4. Select **OK** in the Add Bean secondary window.
5. Move the mouse pointer below the applet outline (the dashed box surrounding the visual beans of the applet). The mouse pointer becomes a cross-hair. Click mouse button 1. A ToDoFile bean called *ToDoFile1* appears on the free-form surface. The free-form surface should look like this:



Adding file dialog beans to the free-form surface

Now that you have added the ToDoFile class as a bean in the free-form surface, the next step is to add file dialog beans for opening files and saving files. Later you will connect these file dialog beans to the **Open To-Do File** and **Save To-Do File** buttons.

These file dialog beans represent standard file dialogs like this one:



File name: *.txt

Files of type: *.*

In the finished To-Do File program, file dialogs will appear when users select the **Open To-Do File** or **Save To-Do File** buttons. In the file dialogs, users will specify the name of the file they want to open or save.

To add the file dialog beans:

1. Select the **Containers** category in the beans palette:

2. Select the **FileDialog** bean:
3. Select the **Sticky** checkbox located just below the beans palette.
4. You want to add the file dialog bean to the free-form surface outside of the applet outline. Move the mouse pointer below the applet outline, to the right of the ToDoFile1 bean, and click mouse button 1. A file dialog bean appears.
5. Move the mouse pointer to the right of the applet outline and click mouse button 1 again. Another file dialog bean appears.
6. Deselect the **Sticky** checkbox.

Now that you have added the file dialog beans, you are ready to customize them. To customize these beans:

1. Change the text of the first file dialog bean to **SaveFile**:

   Select the first file dialog bean, then hold down the Alt key and click mouse button 1 on the text immediately below the bean (it will be something like **FileDialog1**).

   Type *SaveFile*, and then press Shift+Enter.

2. Specify the directory, mode, and title of the SaveFile bean. The mode specifies whether the file dialog is set up for saving or opening ("loading") files. The title is the text that appears in the title bar of the file dialog. To specify these values:

   Select the SaveFile bean and click mouse button 2. The pop-up menu for the bean appears.

   Select **Properties** from the pop-up menu. The Properties secondary window appears.

   Select the field to the right of **directory**. Enter *c:\\* (the two backslashes are an escape sequence that specifies a single backslash).

   Select the field to the right of **mode**. Ensure **SAVE** is selected from the drop-down list.

   Select the field to the right of **title**. Enter *To-Do File to Save*.

3. Following the same procedure that you followed for the SaveFile bean, change the text of the second file dialog bean to *OpenFile*.
4. Specify the following properties for the OpenFile bean:

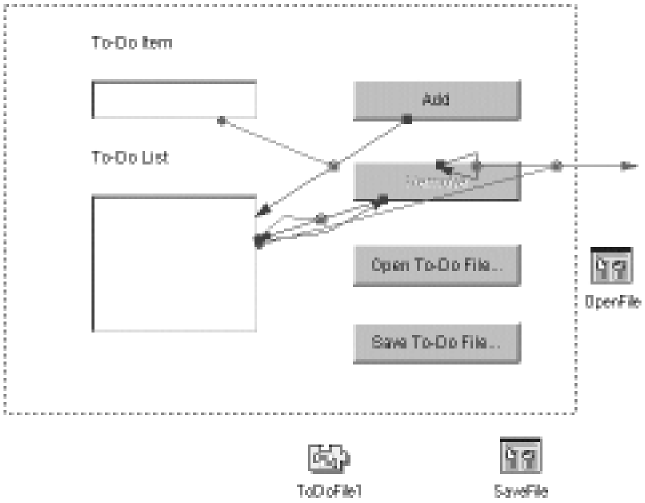
   LOAD as the **mode**

   c:\\ as the **directory**

   .txt* as the **file

   To-Do File to Open as the **title**.

5. Click mouse button 1 on an open area of the free-form surface to confirm your selections. The free-form surface should look like this:



6. Save the current state of your work in the Visual Composition Editor by selecting **Save bean** from the **File** menu.

Note:

> The exact positions of the ToDoFile bean and the file dialog beans do not affect the interface of the finished program. However, it will be easier for you to follow the instructions in the following sections for connecting beans if you line up these three beans according to the instructions in this section.

Connecting the Open To-Do File button

Now that you have added all the new beans to the free-form surface, you are ready to begin connecting them. Let's start with the **Open To-Do File** button.

To begin with, let's list all the actions that the To-Do File program should perform when an end-user selects the **Open To-Do File** button:

1. Show the file dialog.
2. Dispose of the file dialog.
3. Invoke the *readToDoFile* method in the ToDoFile1 bean to read the file that was selected in the file dialog.

You will implement actions 1 and 2 by making connections between the **Open To-Do File** button and the OpenFile file dialog bean. You will implement action 3 by making a connection between the **Open To-Do File** button and the ToDoFile1 bean.

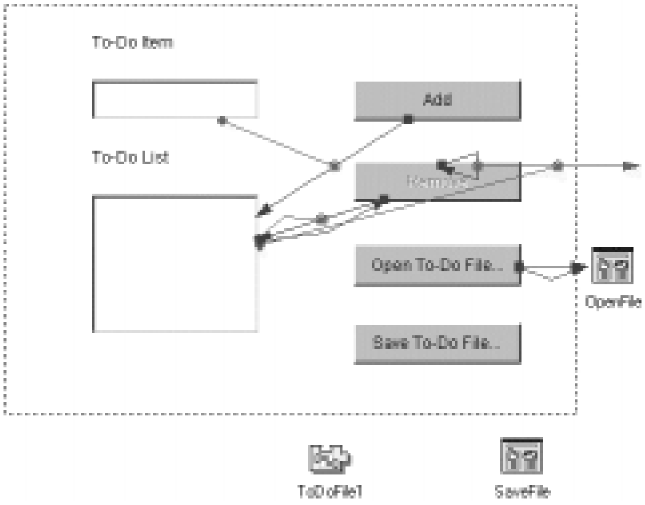
Create the connection to show the file dialog

1. Select the **Open To-Do File** button and click mouse button 2. Select **Connect**, then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears. The mouse pointer changes to indicate that you are in the process of making a connection.
2. Complete the connection by clicking mouse button 1 on the OpenFile bean. From the pop-up menu that appears, select **show().**

Create the connection to dispose of the file dialog

1. Select the **Open To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears.
2. Click mouse button 1 on the OpenFile bean. From the pop-up menu that appears, select **dispose().**

Now the free-from surface should look like this:



You have completed all the connections between the **Open To-Do File** button and the OpenFile bean. Now you are ready to make the connection that invokes *readToDoFile* in the ToDoFile1 bean.

Note:

> You may wonder why the **Open To-Do File** button both shows and disposes of the file dialog. Here's a brief synopsis of what happens. First, notice that there are two connections that both have *actionPerformed(java.awt.event.ActionEvent)* as the source event. One connection has the *show()* method as its target, and the other has the *dispose()* method as its target. Once the *show()* method is called in the file dialog, the file dialog has control until the user selects the **Open** button or the **Cancel**

button. After the user has selected one of these buttons, control returns to the ToDoList class, and the next action it takes is to hide the file dialog by calling its *dispose()* method.

Create the connection to invoke readToDoFile

1. Select the **Open To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears.
2. Click mouse button 1 on the ToDoFile1 bean. From the pop-up menu that appears, select **All Features**.
3. In the **method** list, select **readToDoFile(java.lang.String, java.lang.String, java.awt.List)**, then select **OK**. The connection that appears is incomplete because *readToDoFile* takes three parameters: a directory name, a file name, and a List object. Begin by specifying the directory name:

   Select the connection and click mouse button 2.

   Select **Connect** then **dirName** from the pop-up menu that appears. Notice the selections under **Connect** include the names of all the parameters that you specified for *readToDoFile* when you created it as a method in the ToDoFile class.

   Move the mouse pointer to the OpenFile bean and click mouse button 1. Select **All Features** from the pop-up menu that appears.

   In the **method** list, select **getDirectory()**, then select **OK**.
4. You have specified one of the parameters, but the connection is still not complete. To specify the file name:

   Select the connection between the **Open To-Do File** button and the ToDoFile1 bean and click mouse button 2.

   Select **Connect** then **fileName** from the pop-up menu that appears.

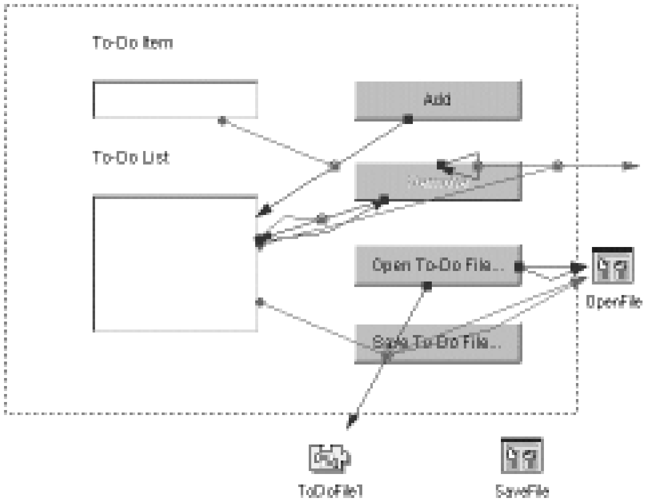
   Click mouse button 1 on the OpenFile bean. Select **All Features** from the pop-up menu that appears.

   In the **method** list, select **getFile()**, then select **OK**.
5. There is still one parameter to specify before the connection is complete: the List object.

   Select the connection between the **Open To-Do File** button and the ToDoFile1 bean and click mouse button 2.

   Select **Connect** then **fillList** from the pop-up menu that appears.

   Click mouse button 1 on the List bean in the applet and select **this** from the pop-up menu that appears. This last connection is significant. It specifies that the List bean in the user interface is the fillList parameter for *readToDoFile*. In other words, the List bean in the interface is the List object in which *readToDoFile* adds items as it reads the input file.
6. The free-form surface should look like this:



Congratulations! You have completed all the connections from the **Open To-Do File** button. Now you are ready to test the work you have done so far on the To-Do File program.

Testing the Open To-Do File button

Now that you have made all the connections for the **Open To-Do File** button, you are ready to test the work you have done so far.

To test the current state of the To-Do File program:

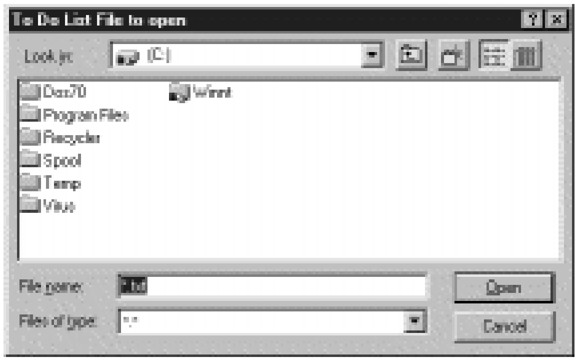
1. First, prepare a simple text file to use for testing. Use the Scrapbook window to create and save a sample To-Do file called `test1.txt` with the following four lines in it:

   ```
   test item 1
   test item 2
   test item 3
   end of test 1
   ```

2. Save your current work in the Visual Composition Editor by selecting **Save bean** from the **File** menu. VisualAge for Java generates code to implement the connections you specified in the last step.
3. Make the Workbench the current window by selecting **Workbench** from the **Window** menu.
4. Ensure the **Projects** page is selected. Find the class ToDoList. You can follow the same process you followed in “Finding your To-Do List applet in the Workbench” on page 23.
5. Select the ToDoList class. Select **Run** then **Run Main** from the **Selected** menu. Note that you have to run the To-Do File program as an application. If you try to run the program as an applet in the applet viewer, you will get an exception as

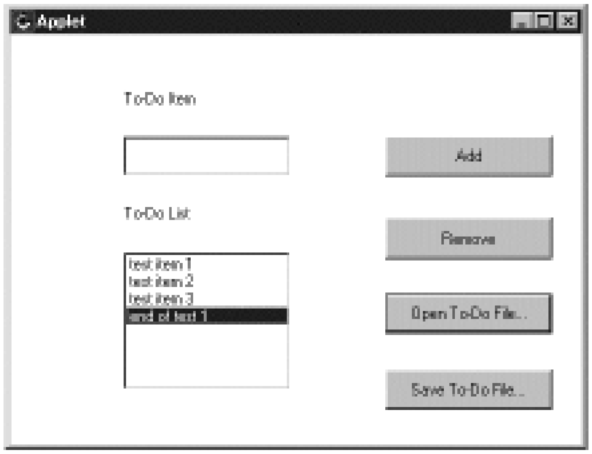
soon as the program tries to open or save a file. Applets are not allowed to access the local file system.

6. The Command Line Argument secondary window appears. Select **Run**. The To-Do File program appears.
7. Select the **Open To-Do File** button. A file dialog that looks like this should appear:



File name: *.txt
Files of type: *.*

8. From this file dialog, go to the directory where you created the `test1.txt` file. Select this file and select **Open**.
9. The **To-Do List** in your program should now be loaded with the items from the `test1.txt` file:



test item 1
test item 2
test item 3
end of test 1

Now that you have tested your current progress on the To-Do File program, you are ready to complete the program by making the connections from the **Save To-Do File** button.

Connecting the Save To-Do File button

You are now ready to make the final connections from the **Save To-Do File** button.

To begin with, let's list the actions that the To-Do File program should perform when the **Save To-Do File** button is selected:

1. Show the file dialog.
2. Dispose of the file dialog.
3. Invoke the *writeToDoFile* method in the ToDoFile1 bean to write the file that was selected in the file dialog.

You will implement actions 1 and 2 by making connections between the **Save To-Do File** button and the SaveFile file dialog bean. You will implement action 3 by making a connection between the **Save To-Do File** button and the ToDoFile1 bean.

As you complete the connections listed in this section, you will notice that they are very similar to the connections you made from the **Open To-Do File** button.

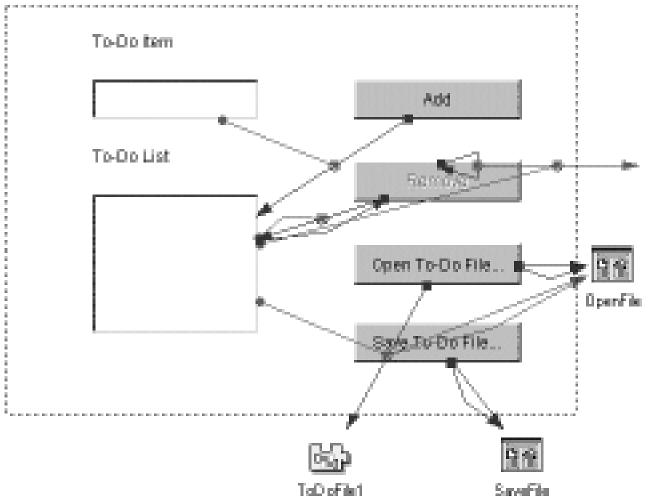
Create the connection to show the file dialog

1. Select the **Save To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears.
2. Click mouse button 1 on the SaveFile bean. From the pop-up menu that appears, select **show().**

Create the connection to dispose of the file dialog

1. Select the **Save To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears.
2. Click mouse button 1 on the SaveFile bean. From the pop-up menu that appears select **dispose()**.

Now the free-form surface should look like this:



You have completed all the connections between the **Save To-Do File** button and the SaveFile bean. Now you are ready to make the connection that invokes *writeToDoFile* in the ToDoFile1 bean.

Create the connection to invoke writeToDoFile

1. Select the **Save To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the pop-up menu that appears.
2. Click mouse button 1 on the ToDoFile1 bean. From the pop-up menu that appears, select **All Features**.
3. In the method list, select **writeToDoFile(java.lang.String, java.lang.String, java.awt.List)**. The connection that appears is incomplete because *writeToDoFile* takes three parameters: a directory name, a file name, and a List object. Begin by specifying the directory name:

   Select the connection and click mouse button 2.

   Select **Connect** then **dirName** from the pop-up menu that appears. Notice the selections under **Connect** include the names of all the parameters that you specified for *writeToDoFile* when you created it as a method in the ToDoFile class.

   Move the mouse pointer to the SaveFile bean and click mouse button 1. Select **All Features** from the pop-up menu that appears.

   In the **method** list, select **getDirectory()**, and select **OK**.
4. To specify the file name:

   Select the connection between the **Save To-Do File** button and the ToDoFile1 bean and click mouse button 2.

   Select **Connect** then **fileName** from the pop-up menu that appears.

Click mouse button 1 on the SaveFile bean. Select **All Features** from the pop-up menu that appears.

In the **method** list, select **getFile()**, then select **OK**.

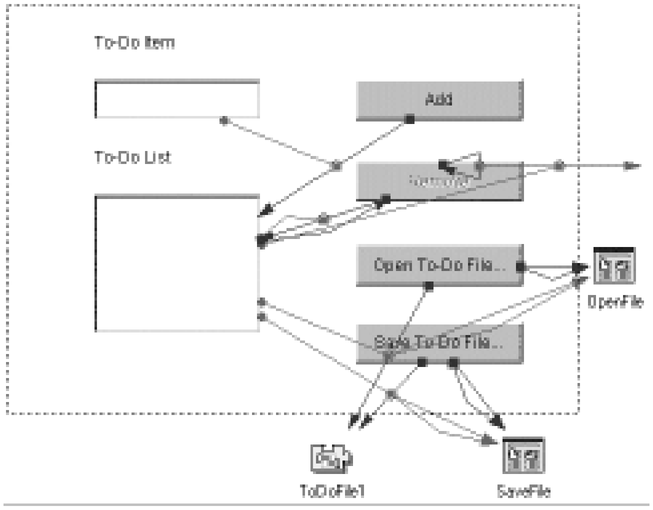
5. There is still one parameter required before the connection is complete. To specify the List object:

   Select the connection between the **Save To-Do File** button and the ToDoFile1 bean and click mouse button 2.

   Select **Connect** then **fillList** from the pop-up menu that appears.

   Click mouse button 1 on the List bean and select **this** from the pop-up menu that appears.

6. The free-form surface should look like this:



Congratulations! You have completed all the connections from the **Save To-Do File** button. Your To-Do File program is complete and you are ready to test it.

Note:

> If you want to do some more work with the Visual Composition Editor, you can try to update the current state of your program to extend the state checking on the **Remove** button. To make this state checking complete, the program needs to check if an item is selected in the List after the **Open To-Do File** button has been selected. Examine the completed version of the To-Do File program if you need some hints before you start. See “Examining examples in the repository” on page 87 for details on how to examine the completed versions of this program.

Saving and testing the completed To-Do File program

Now that you have completed the To-Do File program, you are ready to save and test it.

To save and test your completed To-Do File program:

1. Select **Save bean** from the **File** menu to save your changes. VisualAge for Java generates the code to implement all the work you have done in the Visual Composition Editor since the last time you saved.
2. Show the Workbench window by selecting **Workbench** from the **Window** menu.
3. Select the ToDoList class and click mouse button 2. In the pop-up menu that appears, select **Run** then **Run main**.
4. Try creating and saving a new To-Do file:

   Add the following items to the **To-Do List**. For each item, enter the item in the **To-Do Item** field and select **Add**:

   – *final test item 1*

   – *final test item 2*

   – *final test item 3*

   – *final test item 4*

   – *end of final test*

   Select **Save To-Do File**. A save file dialog should appear. In this dialog, go to the directory where you saved the `test1.txt` file for testing the **Open To-Do File** button. Enter the file name `test2.txt` and select **Save**.
5. Now try loading the list from `test1.txt`. Select **Open To-Do File**. An open file dialog should appear. Select `test1.txt`, then select **Open**. The original list from `test1.txt` should be loaded into **To-Do List**.
6. Now try replacing the current list with the one you saved in `test2.txt`. Select **Open To-Do File**. Select `test2.txt`, then select **Open**. The list from `test2.txt` should replace the `test1.txt` list in **To-Do List**.

Congratulations! You have completed a Java program that combines a user interface created in the Visual Composition Editor with nonvisual code that you created directly.

Before you continue, create a versioned edition of both the ToDoList class and the ToDoFile class. Begin with the ToDoList class:

1. Select the ToDoList class in the Workbench. Select **Version** from the **Selected** menu. The Versioning Selected Items SmartGuide appears.
2. Ensure that **Automatic** is selected and select **Finish**.

Now perform the same steps on the ToDoFile class.

Chapter 5. What else you can do with the Visual Composition Editor

Introduction to what else you can do with the Visual Composition Editor

In Chapter 2, “Building your first applet” on page 9, you learned a great deal about constructing user interfaces using the Visual Composition Editor's beans palette, tool bar, and free-form surface. To build on these fundamental tasks, you need to learn about manipulating beans and their properties, working with connections and their properties, and correcting mistakes.

While reading through this section, you might want to create a new applet, open a Visual Composition Editor on it, and try out some of the tasks described.

Manipulating beans

After you add beans to an applet, you will often want to align them, or size them, or perform similar tasks. Before you can align or size your beans, however, you must learn to manipulate them. This section introduces you to the following tasks:

Selecting beans

Deselecting beans

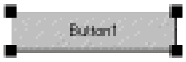
Moving beans

Copying beans

Selecting beans

To select a bean, click on it with mouse button 1.

When you select a bean, small, solid boxes called **selection handles** appear in the corners of the bean to assist you in manipulating that bean.



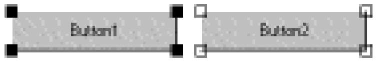
Note:

> Beans that cannot be sized do not have selection handles. Instead, these beans change their background color when they are selected. Beans with this behavior include nonvisual beans and menu beans.

If other beans are selected when you select a bean, they will be deselected automatically. This is referred to as **single selection**. The name of the bean currently selected is displayed in the information area at the bottom of the Visual Composition Editor.

Selecting several beans

If several beans are selected, the last one selected has solid selection handles indicating that it has **primary selection**. The other selected beans have hollow selection handles.



The bean with primary selection is important when performing operations such as bean sizing and alignment. For these operations, the bean with primary selection is the **anchor** for other selected beans. The other selected beans set their alignment or size to the alignment or size of the anchor bean.

To select several beans do one of the following:

- Click mouse button 1 on one of the beans you want to select, then hold down the Ctrl key and click mouse button 1 on each additional bean you want to select. Remember, the last bean selected becomes the anchor around which sizing and alignment operations take place.
- In OS/2, you can click and hold mouse button 1 on a bean. Move the mouse pointer over each additional bean you want to select. After you have selected all the beans you want, release mouse button 1.

When multiple beans are selected, the status area displays ***Multiple selection***.

Deselecting beans

To deselect all the beans currently selected, click mouse button 1 on another bean or in an open area of the free-form surface.

To deselect one bean from a group of beans that have been selected, hold down the Ctrl key and click with mouse button 1 on the bean you want to deselect. If the bean you deselected was the anchor bean, the previously selected bean will become the anchor bean.

Moving beans

To move beans, follow these steps:

1. Click and hold with the appropriate mouse button on the bean.

   In OS/2, hold down mouse button 2 to move beans. In Windows, hold down mouse button 1 to move beans.
2. Move the mouse pointer to the location where you want to position the bean and release the mouse button.

You can move several beans at once by first selecting all of the beans you want to move. You can then **grab** any selected bean (by clicking on it with mouse button 1) and drag all the selected beans to their new location.

Copying beans

After you add a bean, you can copy that bean instead of adding another one from the beans palette. Copying a bean is one method of adding multiple copies of the same bean (using the **Sticky** option from the beans palette is another method). One obvious advantage to copying a bean is that you can make common modifications to one bean and simply duplicate it as often as needed. Copying a bean that has connections does not duplicate the connections.

To copy a bean, follow these steps:

1. Hold down the Ctrl key and select with the appropriate mouse button on the bean you want to copy.

   In OS/2, use mouse button 2 to copy beans. In Windows, use mouse button 1.

2. Drag the mouse pointer to the position where you want the new bean and release the mouse button and the Ctrl key.

You can copy several beans at once by first selecting all the beans you want to copy. Then, press the Ctrl key and grab any selected bean and drag a copy of the beans to their new location.

Copying beans using the clipboard

To copy beans using the clipboard, follow these steps:

1. Select the bean or beans you want to copy.
2. From the **Edit** menu of the Visual Composition Editor, select **Copy**.
3. Then, from the **Edit** menu, select **Paste**. The mouse pointer becomes a cross-hair.
4. Move the mouse pointer to the location where you want to add the new bean or beans and click mouse button 1.

Deleting beans

To delete a bean, simply select it and press the Delete key or select **Delete** from the bean's pop-up menu.

To delete several beans, multiple-select the beans you want to delete prior to performing the delete operation.

If you delete a bean that has connections to or from it, the bean and all of its connections are deleted. However, in this case you are prompted to confirm whether you want to continue before the beans and connections are deleted. If you accidentally delete an item you wish to retain, simply select **Undo** from the **Edit** menu of the Visual Composition Editor.

Sizing, aligning, and positioning beans

This section describes the facilities available in the Visual Composition Editor for sizing, aligning, and positioning beans.

Note:

> One of the properties of an Applet bean is **layout**. You can select a variety of layouts for an applet, but if your applet has a layout other than <**null**>, the sizing, aligning, and positioning facilities described in this section are not available.

Sizing beans

To size a bean, follow these steps:

1. Select the bean you want to size. The selection handles display at each corner.
2. Drag any one of the selection handles using mouse button 1 to adjust the size of the bean.

   Before you release the mouse button, an outline of the bean is displayed to show you the new size of the bean.

To size the bean only horizontally or only vertically, hold down the Shift key while you drag a selection handle in a horizontal or vertical direction.

You can also use the **constraints** property in the bean's Properties window to size the beans. For more information about Properties windows, see “Changing bean properties” on page 62.

Aligning beans

To align beans with other beans, follow these steps:

1. Select the beans you want to align, ensuring the last bean selected is the bean you want the others to align with.
2. Select one of the following alignment tools from the tool bar:

   Align Left

   Align Top

   Align Center

   Align Middle

   Align Right

   Align Bottom

Matching the dimensions of another bean

You can size beans to the same width or height as another bean.

1. Select the beans you want to match, ensuring the last bean selected is the one you want the others to match.
2. Select one of the following sizing tools from the tool bar:

   Match Width

   Match Height

You can also match the dimensions of two or more beans by selecting them and then clicking mouse button 2. Select **Layout** then **Match Size** from the pop-up menu that appears. You can select to match **Width**, **Height**, or **Both**.

Distributing beans evenly

To distribute beans evenly within a **composite** bean (typically an Applet bean), follow these steps:

1. Select the beans you want to distribute evenly.
2. Select one of the following distribution tools from the tool bar:

   Distribute Horizontally

   Distribute Vertically

To evenly distribute beans within an imaginary bounding box that surrounds the multiple-selected beans, follow these steps:

1. Multiple-select the beans you want to evenly distribute. A minimum of three beans must be selected.
2. From the pop-up menu of one of the selected beans, select **Layout Distribute**, and then select one of the following:

   Horizontally In Bounding Box Evenly distribute the selected beans within the area bounded by the left-most edge of the left-most bean and the right-most edge of the right-most bean.

   Vertically In Bounding Box Evenly distribute the selected beans within the area bounded by the top-most edge of the top-most bean and bottom-most edge of the bottom-most bean.

There are two more selections in **Layout Distribute**:

Horizontally In Surface Distributes the selected beans in the same way as **Distribute Horizontally** from the tool bar.

Vertically In Surface Distributes the selected beans in the same way as **Distribute Vertically** from the tool bar.

Changing bean properties

A Properties window provides a way to display and set the properties and other options associated with a bean or connection. In addition to bean-specific properties, you can set data validation and layout properties.

Opening the Properties window for a bean

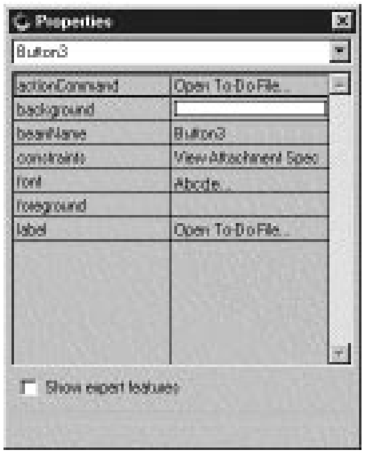
To open the Properties window for a bean, do any of the following:

- Double-click on the bean.
- Select **Properties** from the pop-up menu for the bean.
- Select the bean and select **Properties** from the tool bar:

If you open the Properties window for a bean, you can show the properties of another bean in the window by:

- Selecting another bean
- Selecting another embedded bean from the drop-down list at the top of the Properties window

Here is an example of the Properties window for a bean:



Button3

actionCommand	Open To-Do File...
background	
beanName	Button3
constraints	View Attachment Spec
font	Abcde...
foreground	
label	Open To-Do File...

□ Show expert features

Bean property names and their values are displayed in a table format. How property values are changed depends on the property type itself. For a TextField bean, for example, the value of the *beanName* property is a string and can be changed directly within a cell in the Properties window. Some property values can be changed by selecting from a drop-down list. Other bean property values can be changed through a second window displayed for that purpose. Editing a bean's *background* and *foreground*, for instance, is carried out through the use of the Colors window.

To edit any bean property, open its Properties window and click on the value you want to change. If the value is a string or integer value, you can edit it directly. If the value is a color value, select the ... button in the value column to bring up the Colors window. If the value is a boolean, click on the cell in the value column of the table and select either **True** or **False** from the drop-down list.

After changing the properties of a bean, you can apply them in the following ways:

By selecting another entry in the Properties window

By closing the Properties window

By clicking on another bean or on the free-form surface.

Changing bean colors and fonts

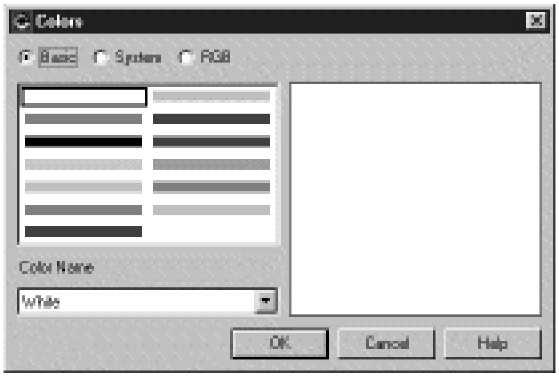
Another enhancement that you can make to your visual beans is to change the colors and fonts that the beans use.

If you are developing applets to be used on multiple platforms, you should carefully consider the effect of choosing colors and fonts that are different from the default system colors and fonts. For example, if you choose a particular font available in OS/2, that font might not be available in Windows. For more information, see “Portability of colors and fonts” on page 65.

Changing the color of a bean

1. In the Visual Composition Editor, double-click on the bean whose color you want to change. The Properties window appears.
2. To change the background color of a bean, select the value for the **background** property in the Property window. Select the ... button that appears.

   The Colors window opens:



● Basic ○ System ○ RGB

Color Name: White

3. In the Colors window, click mouse button 1 on the color you want to use. The name of the color selected is shown in the **Color Name** drop-down list near the bottom of the Colors window. Then select **OK**.
4. To have the change take effect, close the bean's Properties window.

To change the foreground color of a bean, follow the same steps but select the **foreground** property rather than the **background** property in step 2.

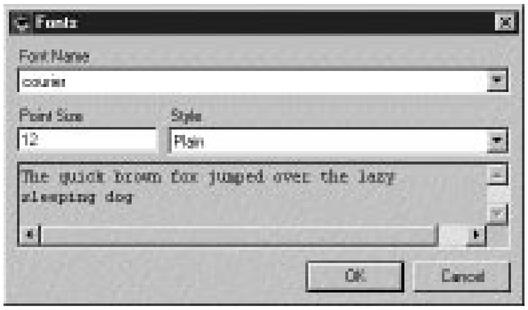
Note:

You cannot change the color of beans in the Menus category.

Changing the font of a bean

1. In the Visual Composition Editor, double-click on the bean whose font you want to change.
2. In the Properties window that opens, select the value of the **font** property. Select the button that appears in the value column for **font**.

   The Fonts window opens:



Font Name: courier

Point Size: 12

Style: Plain

The quick brown fox jumped over the lazy sleeping dog

3. Using the **Font Name** drop-down list, select the font you want to use.
4. Using the **Point Size** and **Style** choices, select the size and style you want to use.

   A sample of the font you have selected is displayed in the text area. You can type additional text in this area to see the appearance of various characters.
5. When you have finished specifying the font, select **OK**. The selected font is shown in the value column for **font**.
6. To have the change take effect, close the bean's Properties window.

Note:

Some beans, such as those in the Menus category, may not support the changing of fonts depending on the target platform.

Portability of colors and fonts

If your applet will be used on multiple platforms, the colors and fonts of the beans must be available on all systems that will run your applet.

If you do decide to change the colors of beans in your applet, use only the basic colors on the Colors window, since non-basic colors may appear differently on different platforms.

If you decide to change the font of a bean, ensure that the font you choose will be available on all the systems that will be running the finished program. You might also have problems with certain fonts if your applet will be run on systems that use code pages designed for languages other than English.

Connecting beans

In Chapter 2, “Building your first applet” on page 9, you learned about making connections. In this section, you explore the different types of connections and what you can do with them. It is best to follow along in the Visual Composition Editor as the different connection types are described and to try any examples discussed. Creating and experimenting with connections is an excellent way to learn how to use them.

Note:

> In “Property-to-property connections” on page 66 you create a new applet. You can reuse this applet to follow along with all of the examples in this section.

There are six types of connections:

Property-to-property Property-to-property connections link two data values together so that if the source and target events are specified in the connection's Property window, when one value changes the other value changes too.

Event-to-method Event-to-method connections call a method when an event occurs.

Event-to-script Event-to-script connections run a script when an event occurs.

Parameter-from-property Parameter-from-property connections use the value of a property as the parameter to a connection.

Parameter-from-script Parameter-from-script connections run a script when a parameter to a connection is required.

Parameter-from-method Parameter-from-method connections use the result of a method as a parameter to a connection.

Event-to-script and parameter-from-script connections enable you to connect to non-public methods of the composite bean.

A connection has a **source** and a **target**. The point at which you start the connection is called the source. The point at which you end the connection is called the target. For information on connection properties, see “Changing the properties of connections” on page 71.

Note:

> If a particular bean method, property, or event does not appear in the bean's preferred connection list in its bean or connection pop-up menu, you can select **Connect** then **All Features** from the bean pop-up or connection pop-up menu to display a complete list. The list of methods, properties, and events displayed in a window opened by selecting **All Features** represents a bean's complete **public interface**.

Property-to-property connections

Property-to-property connections tie two data values together. The color of this connection type is blue. A simple example of a property-to-property connection follows:

1. Create a new applet using the Create Applet SmartGuide:

   Select **Create Applet** from the Workbench tool bar:

   In the Create Applet SmartGuide, enter a name for the applet in **Name of Applet** and specify a project and package for the applet. Ensure **Design the applet visually** is selected and select **Finish**.

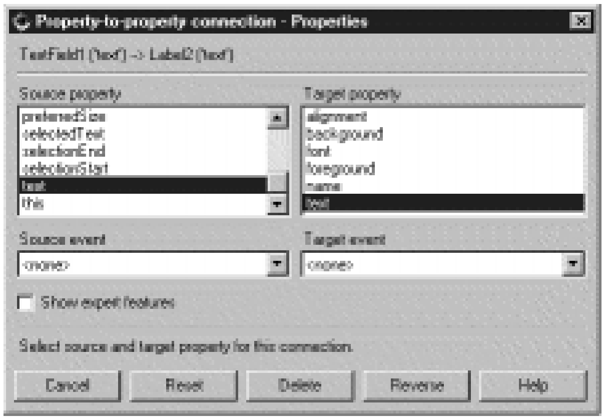
2. When the Visual Composition Editor opens on your new applet, place a TextField bean and a Label bean within the default Applet bean.

3. Connect the *text* property of the TextField bean to the *text* property of the Label bean:

   Select the TextField bean and click mouse button 2. Select **Connect** then **text** from the pop-up menu that appears.

   Click mouse button 1 on the Label bean and select **text** from the pop-up menu that appears.

4. Select the new connection you just created and click mouse button 2. Select **Properties** in the pop-up menu that appears. The Property-to-property connection Properties window appears.


Property-to-property connection - Properties

TextField1 (text) -> Label2 (text)

Source property: preferredSize, selectedText, selectionEnd, selectionStart, text, this

Target property: alignment, background, font, foreground, name, text

Source event: <none>

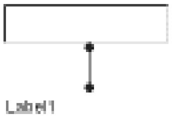
Target event: <none>

□ Show expert features

Select source and target property for this connection.

5. In the Properties window, select **textValueChanged(java.awt.event.TextEvent)** for the **Source event** and select **OK**.

The free-form surface should look like this:



When you run the applet that contains these beans, the Label is updated every time you change the text in the TextField.

For property-to-property connections, either endpoint can serve as the source or target. The only time it matters which property is the source and which is the target for a connection is at initialization. During initialization, the value of the target is updated to match the value of the source.

A property-to-property connection is initiated from the source bean's **Connect** choice in the pop-up menu and is terminated by selecting the appropriate target bean's property.

Event-to-method connections

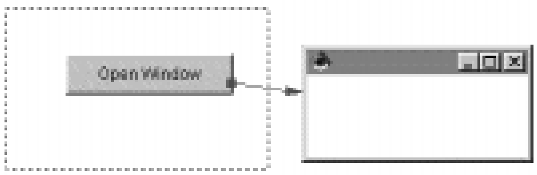
Event-to-method connections cause a method to be called when a certain event takes place. The color of this connection type is green.

For event-to-method connections, the event is always the source and the method is always the target. If you try to connect a method to an event, VisualAge for Java automatically reverses the source and target for you, so the connection is an event-to-method. A simple example of an event-to-method connection follows:

1. Place a Button bean within the default Applet bean in the Visual Composition Editor. Change the text of this button to *Open Window*.
2. Place a Frame bean from the **Containers** category on the free-form surface of the Visual Composition Editor.

3. Connect the *actionPerformed(awt.java.event.ActionEvent)* event of the Button to the *show* method of the Frame bean. This connection causes a frame to display when the Button is selected.

The free-form surface should look like this:



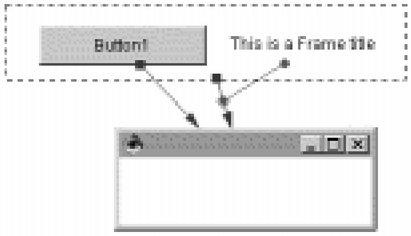
Properties can also be used to call a method, and events can be used to change the value of a property. This behavior is possible because VisualAge for Java can associate an event with a change in property value. As a result, you can make the following connections with properties:

Connecting an event to a property

In addition to calling a method, an event can also be used to set a property value. In this case, a parameter must be used with the connection to supply the property value. A simple example of an event-to-property connection follows:

1. Place a Label bean within the default Applet bean in the Visual Composition Editor and change its *text* property in the Properties window to the string *This is a frame title*.
2. Place a Button bean within the default Applet bean in the Visual Composition Editor.
3. Place a Frame bean on the free-form surface of the Visual Composition Editor.
4. Connect the *actionPerformed(awt.java.event.ActionEvent)* event of the Button to the *show* method of the Frame bean.
5. Connect the *componentShown(java.awt.event.ComponentEvent)* event of the default Applet bean to the *title* property of the Frame bean.
6. Now, to provide the parameter for the event-to-property connection you just created, connect the *text* property of the Label bean to the *value* property in the connection pop-up menu.

The free-form surface should look like this:



When you run the applet and select the button, the title text of the frame will be set to *This is a frame title*. This example is rather contrived, but it conveys the idea. For more information, see Connection parameters.

Event-to-script connections

Event-to-script connections run a given method when a certain event takes place. This provides a way to implement or alter applet behavior directly through the use of the Java language. The target of an event-to-script method can be any method in the class that you are manipulating in the Visual Composition Editor.

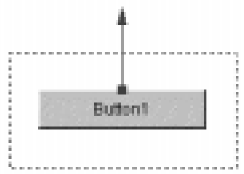
Note:

> An event-to-method connection is made between two beans. An event-to-script connection is made between a bean and a method in the composite bean. The method in the composite been does not have to be public.

The color of an event-to-script connection is green. To create an event-to-script connection:

1. Select the source bean (for example, a Button). Click mouse button 2 to display the bean's pop-up menu. Select **Connect** then select an event, such as **actionPerformed(java.awt.event.ActionEvent)**. The mouse pointer changes.
2. Move the mouse pointer to any open area of the free-form surface. It cannot be over any bean, including the default Applet bean. Click mouse button 1 and select **Event to Script** from the pop-up menu.
3. The resulting window allows you to pick from the list of available methods.
4. Once you've selected a method, select **OK** to complete the connection.

The connection is drawn between the source bean and the outer edge of the Visual Composition Editor free-form surface.



Parameter connections

The last three types of connections supply a parameter to a connection from various sources:

1. Parameter-from-property
2. Parameter-from-script
3. Parameter-from-method

The color for a parameter connection line is purple.

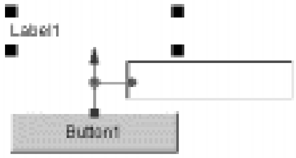
Parameter-from-property

Parameter-from-property connections use the value of a property as the parameter to a connection. As with other connection types, a parameter-from-property connection is initiated from the source bean's **Connect** choice in the pop-up menu and is terminated by clicking mouse button 1 over the target connection line requiring the parameter. The appropriate property is then selected from the pop-up menu for the connection.

A parameter-from-property connection was used in building the To-Do List sample. When we connected the TextField bean's *text* property to the connection between the Button bean and the List bean, we were making a parameter-from-property connection. Refer to “Connecting beans” on page 17 for the specific connection details for the To-Do List sample. The following example also illustrates the use of the parameter-from-property connection:

1. Place a Label bean, a TextField bean, and a Button bean within the default Applet bean in the Visual Composition Editor.
2. Connect the *actionPerformed(awt.java.event.ActionEvent)* event of the Button bean to the *text* property of the Label bean.
3. Now, make the parameter-from-property connection by connecting the *text* property of the TextField bean to the *value* property of the connection pop-up menu.

The free-form surface should look like this:



When you run the applet and type text into the text field and select the button, the label text is set to match the text in the text field.

Parameter-from-script

Parameter-from-script connections run a method whenever a parameter to a connection is required. This connection is much the same as a parameter-from-property connection except that the value supplied to the connection is returned from a Java method instead of a bean property value.

For the sake of illustration, assume that we have created a simple method called *stringFromScript* in the applet class that returns the text *this is a string*. An example of a parameter-from-script connection follows. Unlike other connection types, a parameter-from-script connection is initiated from the parameter name in the **Connect** menu choice in the connection's pop-up menu and is terminated as follows:

1. Place a Label bean and a Button bean within the default Applet bean in the Visual Composition Editor.

2. Connect the *actionPerformed(awt.java.event.ActionEvent)* event of the Button to the *text* property of the Label bean.
3. Now, create the parameter-from-script connection by connecting the *value* property in the connection pop-up menu to the method as follows:
   - Click mouse button 2 on the connection and select **Connect** then **value** from the pop-up menu. The mouse pointer changes.
   - Click mouse button 1 on any open area of the free-form surface and select **Parameter from Script** from the pop-up menu.
   - The resulting window allows you to pick from the list of available methods. In our case, the method *stringFromScript()* appears (this is the method that we created in the applet class).
   - Once you've selected a method, select **OK** to complete the connection.

Note:

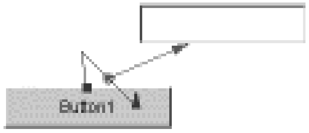
> You have already seen a parameter-from-script connection in Chapter 3, “Adding state checking to your applet” on page 23. The connection you made to the *enableRemove(java.awt.List)* method is a parameter-from-script connection.

Parameter-from-method

Parameter-from-method connections use the result of a method as a parameter to a connection. An example of how to use a parameter-from-method connection follows:

1. Place a Button bean and a TextField bean within the default Applet bean in the Visual Composition Editor.
2. Connect the *actionPerformed(awt.java.event.ActionEvent)* event of the Button bean to the *label* property of the Button bean. Yes, connecting an event to a property for the same bean does make sense in the right situation.
3. Then, make the parameter-from-method connection by connecting the *getText()* method of the TextField bean to the *value* property in the connection's pop-up menu. This provides the needed connection parameter and causes the connection line to become solid in color.

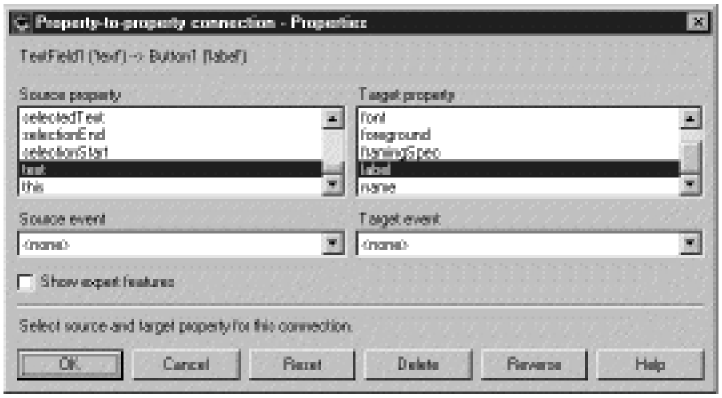
When you test the applet, type a string into the text field and then select the Button. The string becomes the label for the Button.



Changing the properties of connections

Connections, like beans, have properties. To open the properties for a connection, select **Properties** from the connection's pop-up menu. Or, just double-click on the connection.

The following figure shows the Properties window for a property-to-property connection:



Property-to-property connection - Properties

TextField1 ("text") -> Button1 ("label")

Source property: selectedText, selectionEnd, selectionStart, **text**, this

Target property: font, foreground, layoutSpec, **label**, name

Source event: <none>

Target event: <none>

☐ Show expert features

Select source and target property for this connection.

You can use a connection's Properties window to change the source or target property of the connection. To do this, select a different source or target property from the appropriate list. To display the current source and target properties of the connection, select **Reset**.

If you want the source property of a property-to-property connection to be the target property, and vice versa, you can change the source and target properties by selecting **Reverse** in the connection's Properties window.

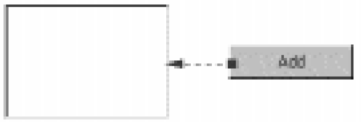
When you have finished changing the connection's properties, select **OK**.

Connection parameters

Event-to-method and event-to-script connections sometimes require parameters (or arguments). The method's parameters are available as properties of the connection. Therefore, to specify a parameter you simply make a connection to the parameter property of the event-to-method connection itself.

When a connection requires parameters that have not been specified, it appears as a dashed line, indicating that the connection is not complete.

In the *To-Do List* applet, you connected the Button bean's *actionPerformed(awt.java.event.ActionEvent)* event to the List bean's *addItem(java.langString)* method, and a dashed line resulted:



The parameters that methods require are indicated by the items inside parentheses () in the method name. For example, the *addItem(java.lang.String)* method takes one parameter, a String. A method named *addItem(java.lang.String, int)* takes two parameters, a String and an int.

Once you have specified all the necessary parameters, the connection line becomes solid, indicating the connection is complete. If you do not supply enough parameters for a connection, the connection continues to appear as a dashed line.

To specify parameters you can use properties or constants.

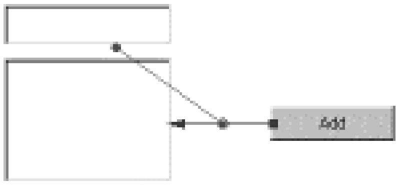
Properties as parameters

Most of the time, the parameters you need are properties of other beans you are working with in the Visual Composition Editor. To use a bean's property as a parameter:

1. Make a new connection using the bean's property as the source.
2. For the target, click mouse button 1 on the connection line that requires the parameter, and then from its connection menu, select the particular parameter property you are specifying.

   While making a connection to a connection line, you will see a small visual cue in the middle of the connection line when the mouse pointer is directly over the connection line, indicating the pointer is positioned correctly:

In the *To-Do List* applet, the text entered in the TextField bean is used as the parameter of the event-to-method connection between the **Add** Button and the List bean.



In this example, you provided the parameter of the event-to-method connection by making a property-to-property connection between the TextField bean's *text* property and the event-to-method connection's *item* property. The connection's *item* property is the name of the first and only parameter of the *addItem (java.lang.String)* method.

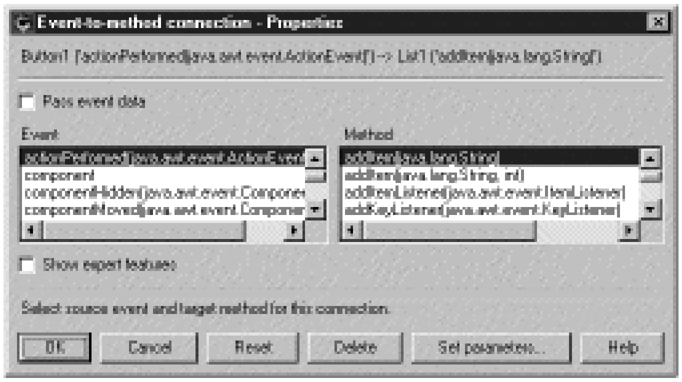
Constants as parameters

Parameter values can also be constants. You specify a constant value for a parameter in the Properties window for the connection.

For example, to specify a constant value for the parameter for an event-to-method connection:

1. Double-click on the event-to-method connection.

The Properties window for an event-to-method connection opens:



Event-to-method connection - Properties

Button1 (actionPerformed(java.awt.event.ActionEvent)) --> List1 (addItem(java.lang.String))

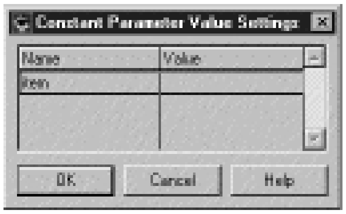
□ Pass event data

□ Show expert features

Select source event and target method for this connection.

Event
actionPerformed(java.awt.event.ActionEvent
component
componentHidden(java.awt.event.Componen
componentMoved(java.awt.event.Componen
Method
addItem(java.lang.String)
addItem(java.lang.String, int)
addItemListener(java.awt.event.ItemListener)
addKeyListener(java.awt.event.KeyListener)
OK
Cancel
Reset
Delete
Set parameters...
Help

2. In this window, select **Set parameters**. The Constant Parameter Value Settings window opens:



Constant Parameter Value Settings

Name	Value
item	

3. In the Constant Parameter Value Settings window, type the constant values for the parameters you want to add.

4. When you finish, select **OK**, and then select **OK** in the Properties window for the event-to-method connection.

Manipulating connections

Like beans, once connections are made, you can manipulate them in many different ways.

Selecting and deselecting connections

You select and deselect connections the same way you select and deselect beans. You can select multiple connections. The information about the currently selected connection is displayed in the information area at the bottom of the Visual Composition Editor.

Note:

> You cannot select beans and connections at the same time.

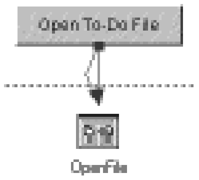
Deleting connections

To delete a connection, select **Delete** from its pop-up menu. You can also delete a connection by selecting the connection and pressing the Delete key.

To delete several connections, select the connections you want to delete, and then select **Delete** from the pop-up menu of one of the selected connections.

Reordering connections

When you make several connections from the same event or property of a bean, the connections run in the order in which they were made. However, if you create the connections in a different order than the order in which you want to run them, you can reorder them. Consider, for example, the To-Do File program. You made two connections between the **Open To-Do File** button and the OpenFile bean: one to show the file dialog, and one to dispose of it.

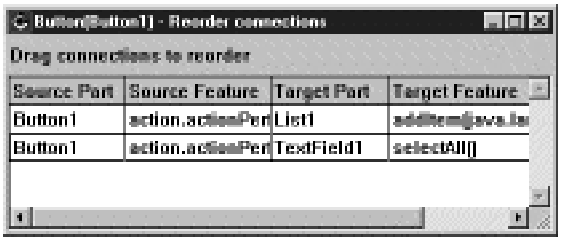


Suppose you made the **dispose** connection first and then the **show** connection. The behavior of the program would be incorrect. However, you could reorder these two connections to make the **show** connection first.

If you need to change the order of connections, simply reorder the connections from the bean by doing the following:

1. From the bean's pop-up menu, select **Reorder Connections From**.

   The Reorder connections window appears:

   Drag connections to reorder

Source Part	Source Feature	Target Part	Target Feature
Button1	action.actionPer	List1	addItem(java.la
Button1	action.actionPer	TextField1	selectAll()

   The Reorder connections window contains all of the connections for the bean you selected. In this example, the *(Button1, actionPerformed(awt.java.event.ActionEvent) -> List1, addItem(java.lang.String))* connection is the first one that runs for the selected bean.

2. In the Reorder connections window use the appropriate mouse button to drag each connection to its appropriate position in the list. In OS/2, use mouse button 2; in Windows use mouse button 1.

   As you drag a connection through the list, a dark line appears to indicate where the connection will be inserted when you release the mouse button.

Showing, hiding, and browsing connections

You can show and hide connections by using the **Hide Connections** tool: and the **Show Connections** tool: from the tool bar. These tools show and hide all connections to and from the selected bean or beans. If no beans are selected, these tools will show and hide all the connections in the Visual Composition Editor.

You can selectively show and hide a bean's connections by selecting **Browse Connections** from the bean's pop-up menu and then selecting one of the following:

Show To Shows all connection lines extending to the bean

Show From Shows all connection lines extending from the bean

Show To/From Shows all connection lines extending to and from the bean

Show All Shows all connection lines

Hide To Hides all connection lines extending to the bean

Hide From Hides all connection lines extending from the bean

Hide To/From Hides all connection lines extending to and from the bean

Hide All Hides all connection lines

Arranging connections

When you select a connection, selection handles are displayed at both ends and along the connection line. You can then drag the mid-point selection handle to a new position. This makes the connection line draw in a different area of the free-form surface, which can help you distinguish among several connection lines that are close together. When additional selection handles appear, you can then drag the middle selection handle to a new position to bend the connection line even further.

To restore a connection line to its original shape, from the pop-up menu for the connection line, select **Restore Shape**.

Changing connection endpoints

VisualAge for Java gives you the ability to change the endpoint bean of a connection, meaning you can change the source or target bean of the connection. It is quicker than deleting the connection and creating a new one.

For event-to-method and property-to-property connections, you can move either end of the connection. For event-to-script connections, you can only move the event end of the connection.

To change either end of a connection:

1. Select the connection whose endpoint you want to change.

   Selection handles appear along the connection line.

2. Move the mouse pointer over the selection handle at the end of the connection you want to change. Using the appropriate mouse button, drag the selection handle to the new bean. In OS/2, use mouse button 2; in Windows use mouse button 1.

   If you change the endpoint of a connection to a bean that does not have the same property available as in the original connection, the bean's connection pop-up menu appears so you can specify a new property to connect to.

Chapter 6. Managing editions

Introduction to managing editions

You've just reached a milestone in the development of your program, and you're ready to start coding some new features. Maybe you just want to explore a different (perhaps more efficient) implementation of a method that already works, but you're not sure if changes or additions will introduce new problems. This is a good time to create a new versioned edition of your code.

With VisualAge for Java, you can manage multiple editions of program elements. You have already seen some of the concepts for managing editions. This section briefly reviews these concepts and shows you how to use the edition management features of VisualAge for Java.

In this section, you'll learn about:

- Editions in VisualAge for Java
- Versioning an edition
- Updating your program with the assurance of easily reverting back
- Returning to a previous edition
- Exploring the repository
- Managing the workspace

About editions

As you've been saving your program elements, VisualAge for Java has been keeping track of your code. In fact, the code you are working on is saved in an edition. An edition is a "cut" or "snapshot" of a particular program element.

To see more information on the edition you're working on, select the **Show Edition Names** tool from the tool bar in the Workbench:

Notice that each program element includes either an alphanumeric name or a timestamp beside it; this is the edition information (described below in more detail). You can also see the same information from the **Source** pane. For example, select your ToDoFile class and move the mouse over the class icon:

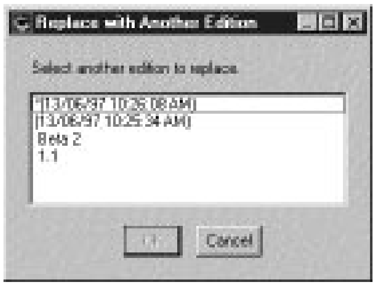
in the **Source** pane title bar. The hover-help window displays the edition information. The edition information is also displayed in the status area below the **Source** pane.

An edition of a program element keeps track of all code within the program element, including program elements within it. For example, an edition of a package includes classes and interfaces and the methods within these classes and interfaces.

An edition of each program element exists in both the workspace and the repository. The center of activity in the VisualAge for Java programming environment is the **workspace**, which contains the Java source code for all your current projects as well as the standard Java class libraries.

To help manage multiple editions of your program elements, VisualAge for Java also includes the **repository**, a storage area that can contain multiple editions of program elements. While the repository is not a development environment, you can explore and retrieve its contents, as needed. You can save as many editions of a program element as you wish. All editions are stored and are accessible from the repository.

The workspace can only hold one edition of a program element at any given time. This edition is called the current edition. If you replace this edition with another edition from the repository, that new edition becomes the current edition. Note that the current edition is always marked by an asterisk (by default) to the left of the edition name when you browse an edition list in the repository.



Select another edition to replace.

```
*(13/06/97 10:26:08 AM)
(13/06/97 10:25:34 AM)
Beta 2
1.1
```

There are two fundamental types of editions:

- Open edition

  An open edition of a program element can be modified. You can bring this edition into the workspace, making it the current edition, and change it as required. In the screen image above, the open editions are marked by timestamps. For example, (`13/06/97 10:25:34 AM`) is an open edition.

- Versioned edition

  A versioned edition of a program element cannot be changed. When you version an edition, you establish a frozen (read-only) code base to which you can revert any time. In the screen image above, versioned editions are designated by alphanumeric names (for example, `Beta 2` or `1.1`).

A versioned edition can also be the current edition, although any changes you make and save automatically create a new open edition.

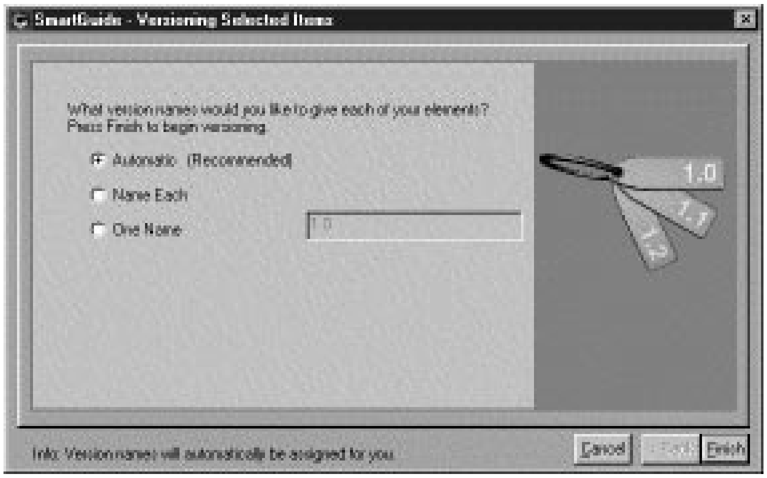
When you save a program element, not only is your code incrementally compiled behind the scenes, the current edition is updated in both the workspace and the repository.

Versioning an edition

You can version a project, a package, or a class. When you version one of these program elements, all program elements contained within it are also versioned. For example, if you version a package, all classes that are part of that package are also versioned.

Let's create a versioned edition of your code.

1. Select the package in which you created your To-Do List applet, then select **Version** from the pop-up menu. The Versioning Selected Items SmartGuide appears.



SmartGuide - Versioning Selected Items

What version names would you like to give each of your elements?
Press Finish to begin versioning

● Automatic (Recommended)

○ Name Each

○ One Name 1.0

Info: Version names will automatically be assigned for you

2. Ensure the Automatic radio button is selected and then select **Finish**.

In the Workbench hierarchy, notice that the timestamp beside the package name has been replaced with the new version number. This versioned edition is now permanently stored in the repository, regardless of what happens to your editions in the workspace. You can create open editions based on this versioned edition, and the versioned edition will always be available from the repository.

Updating your code again

Now that you have a versioned edition of your program in the repository, you can change your program elements in the workspace with the assurance that you can always revert back to the versioned edition.

Creating a new edition

Because a versioned edition cannot be modified, you will need to create a new open edition from the versioned edition before you can continue changing the program element. If the current edition in the workspace is the versioned edition, a new edition is automatically created for you if you make changes to the program element and then save it. For example:

1. Select your ToDoList class in the Workbench, and type a new comment in the **Source** pane.
2. From the pop-up menu in the Source pane, select **Save**.

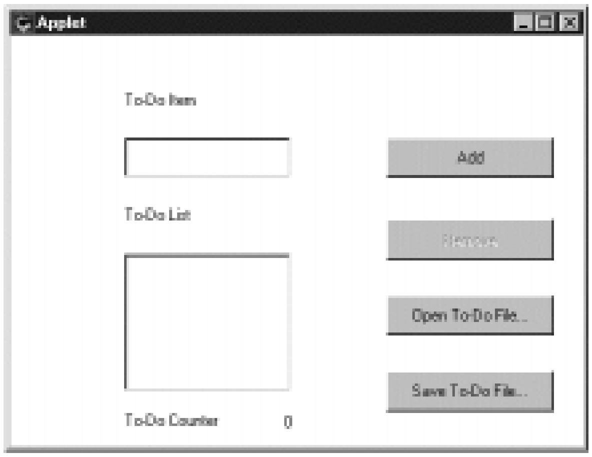
Notice that the edition name (in the hierarchy pane) changes from the versioned edition name to a timestamp. Because the workspace can only hold one edition of a program element at any given time, the new edition replaces the versioned edition. (Of course, a copy of the versioned edition can always be retrieved from the repository.)

Adding a counter to the ToDoFile program

Let's add a counter to the ToDoFile program, which will reflect the number of items in the To-Do list at any given time. To add this feature, we need to change the applet as follows:

1. Add Labels for the counter name and the counter itself.
2. Connect the **Add**, **Remove**, and **Open To-Do File** buttons to the counter Label.

When modified, the applet will look like this:

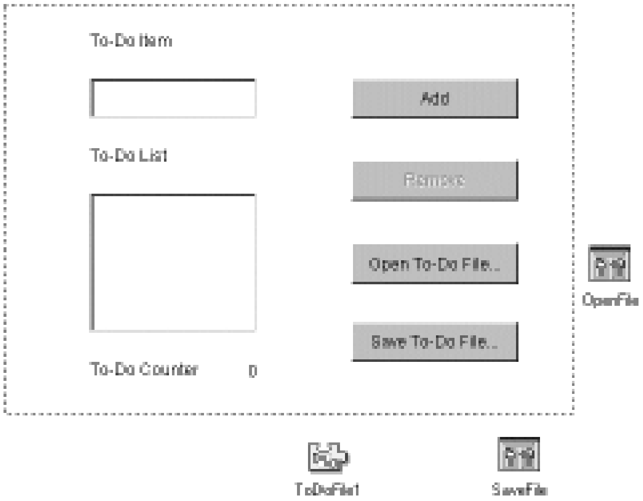


Adding the labels

To add the two Labels using the Visual Composition Editor:

1. Select the ToDoList class in the Workbench and select **Open To** then **Visual Composition** from the **Selected** menu. This opens the ToDoList class in the Visual Composition Editor.
2. To make it easier to create the new connections, hide the existing connections by selecting **Hide Connections** from the tool bar:
3. Select the **Data Entry** category from the beans palette, and select the **Label** bean.
4. Click mouse button 1 beneath the List to add a Label.
5. Modify the text of the Label to **To-Do Counter**.
6. Add another Label to the right of this Label.
7. Double-click on this new Label to open its Properties window. Select the value field to the right of the **alignment** field. From its pull-down menu, select **RIGHT**, which right-justifies the value. In the **text** field, change the value to **0**, which is the initial value of the counter. Close the Properties window.
8. Align the two Labels.
   - Select the counter Label, then the List, and select the **Align Right** tool from the tool bar.
   - Select the counter name Label, then the List, and select the **Align Left** tool.
   - Select the counter name Label, then the counter Label, and then select the **Align Middle** tool.

The visual beans have been added and aligned. The free-form surface should look like this:



Now you're ready to add the connections.

Note:

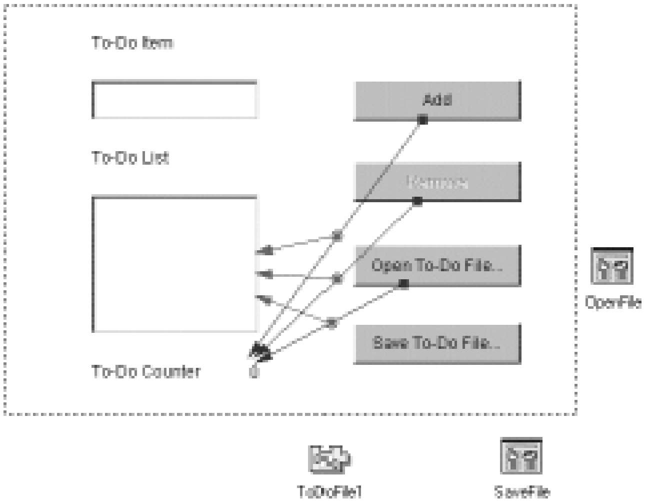
All the other connections you made are still there, they are just hidden now because you selected **Hide Connections** from the tool bar. The new connections that you make in the next step will not be hidden.

Connecting the labels

To connect the **Add** button to the counter:

1. Select the **Add** button and click mouse button 2. From the pop-up menu select **Connect** and then **actionPerformed(java.awt.event.ActionEvent)**.
2. Position the mouse over the counter Label and click mouse button 1.
3. From the pop-up menu, select **text**. A dashed green line now appears, indicating an incomplete connection.
4. Select the connection and click mouse button 2. Select **Connect** and then **value** from the pop-up menu that appears.
5. Position the mouse over the list and click mouse button 1.
6. From the pop-up menu, select **All Features** to bring up the Connect property named window.
7. From the **method** list, select the **getItemCount()** method and then select **OK**. This provides the count of the list as input for setting the counter Label string. The connection is now complete.
8. Connect the **Remove** and **Open To-Do File** buttons in the same manner. You're simply updating the list count in the counter whenever an action is taken that may modify the list count. In this applet, any of the top three buttons have this potential.

Now the free-form surface should look like this:



From the **File** menu, select **Save Bean**. The changes you've made are reflected in this open edition. Select the **Test** tool from the tool bar to launch the applet viewer and see the counter in action.

Returning to a previous edition

Your program now contains a counter. It works fine, but after thinking about it for a while, you decide that you want to keep the interface as clean as possible -- no bells and whistles. So you want to take out the counter code. Of course, you could just delete the labels and connections you've added, but what if you inadvertently delete one of the other program elements or connections? No need to worry. Remember, you versioned the previous edition!

Follow these steps to replace the current edition with a previous edition from the repository:

1. Select the ToDoList class in the Workbench and click mouse button 2.
2. From the pop-up menu, select **Replace With** and then select **Another Edition.**
3. From the **Replace with Another Edition** secondary window, select the edition that you previously versioned and select **OK.** (Because you want to replace the current edition with the previous edition, you could have also selected **Replace With** and then **Previous Edition** from the pop-up menu.)

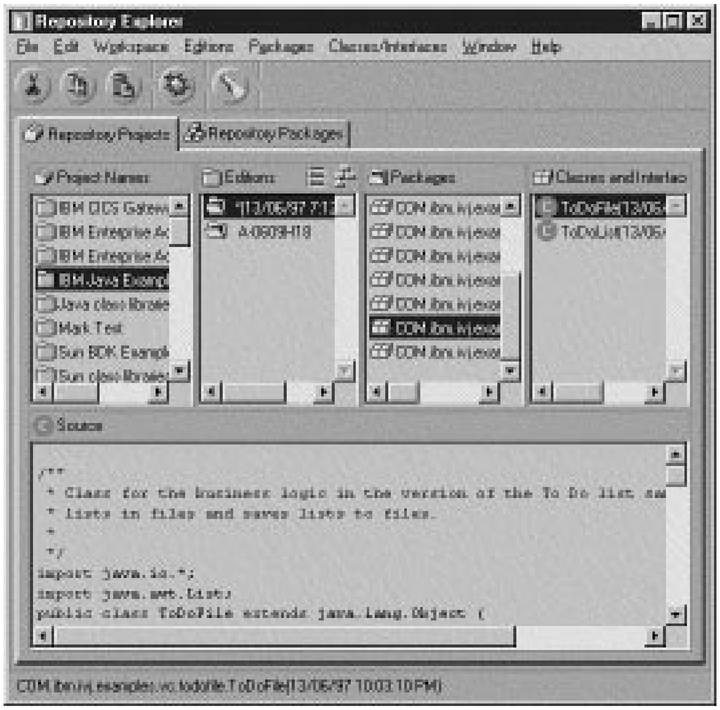
The edition information beside the class name now indicates the version number, not the timestamp of the open edition you had been working on.

If you change your mind again and decide that the counter should stay, you can always add it back; the edition that contained the counter is still in the repository.

Exploring the Repository

More than just a suite of edit-compile-debug tools, VisualAge for Java provides robust code management facilities. You've seen how easy it is to work with multiple editions of a program element. But what else can you get from the repository?

From the **Window** menu, select **Repository Explorer**.



```
/**
 * Class for the business logic in the version of the To Do list sa
 * lists in files and saves lists to files.
 *
 */
import java.io.*;
import java.awt.List;
public class ToDoFile extends java.lang.Object {
```

COM.ibm.ivj.examples.vc.todofile.ToDoFile(13/06/97 10:03:10 PM)

The Repository Explorer provides a visual interface of your repository. The repository includes all editions of all program elements. This includes all the program elements that are currently in the workspace.

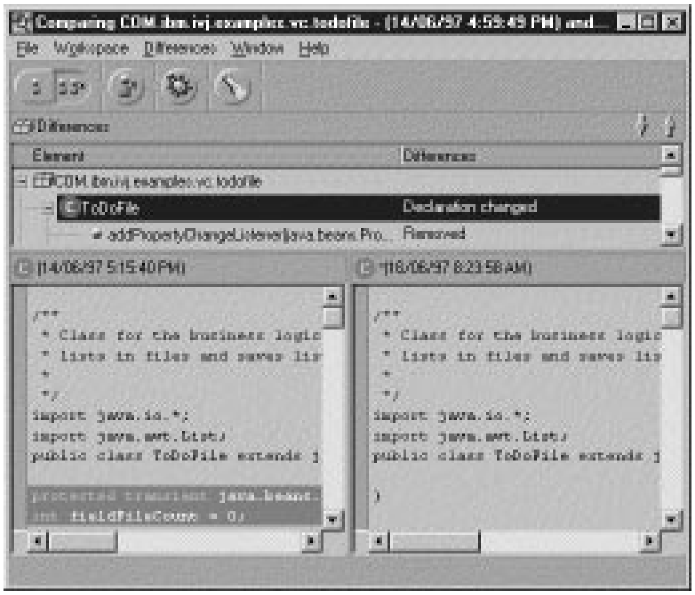
Within the Repository Explorer, you can also open or compare program elements that are stored in the repository. There's no need to swap editions in and out of the workspace to view them or compare them.

By comparing different editions, you can see:

- What changes have been made as a result of code generation
- Precisely how an edition with errors differs from a bug-free edition

To compare two editions of a package:

1. Select the **Repository Packages** page.
2. Select the package in which you created the To-Do List applet from the **Package Names** list.
3. From the **Editions** list, hold down mouse button 1 and drag-select the top two editions. From the pop-up menu, select **Compare**. The Comparing window appears:



Comparing COM.ibm.ivj.examples.vc.todofile - (14/06/97 4:55:45 PM) and...
File Workspace Differences Window Help
Differences
Element
Differences
COM.ibm.ivj.examples.vc.todofile
ToDoFile
Declaration changed
addPropertyChangeListener(java.beans.Pro...
Removed
(14/06/97 5:15:40 PM)
*(16/06/97 8:23:58 AM)

```
/**
 * Class for the business logic
 * lists in files and saves lis
 *
 */
import java.io.*;
import java.awt.List;
public class ToDoFile extends j

protected transient java.beans.
int fieldFileCount = 0;
```

```
/**
 * Class for the business logic
 * lists in files and saves lis
 *
 */
import java.io.*;
import java.awt.List;
public class ToDoFile extends j

}
```

4. Select a class or method name in the **Element** pane, and you'll see two sets of corresponding code in the text panes below. Here, you can compare the two program elements. From the **Differences** pull-down menu, you can select **Next Difference** and **Previous Difference**. You can also select the arrows: ↓ ↑ in the upper right corner of the window to move back and forth in the list of differences.

All program elements that are in the workspace are indicated by an asterisk (*).

Examining examples in the repository

VisualAge for Java comes with a wide variety of example code. Use the Repository Explorer window to examine these examples. For instance, to examine the completed version of the To-Do File program in the Repository Explorer:

1. Select the **Repository Projects** page. Select **IBM Java Examples** from the **Project Names** list.
2. Select an edition from the **Editions** list and select **COM.ibm.ivj.examples.vc.todofile** from the **Packages** list. The classes in this package appear in the **Classes and interfaces** list. To examine one of the classes in this list, select the class and click mouse button 2. Select **Open** from the pop-up menu that appears.

Suppose that you want to run these completed samples, or make your own updates to them. First, you must bring them into the workspace. To bring the completed version of the To-Do File program, for example, into the workspace:

1. In the Workbench, select the project into which you want to add the To-Do File program and select **Add package** from the **Selected** menu. The Add Package SmartGuide appears.

2. Select **Add package(s) from the repository** and then select **Browse**. The Add Packages from Repository secondary window appears.
3. Select **COM.ibm.ivj.examples.vc.todofile** from the **Available package name** list. Select an edition from the **Editions list** and select: >> to add the edition to the **Editions to Add** list. Select **OK**.

The package for the To-Do File program is added to your workspace, and you can update it and run it.

Summary

With the repository and the ability to work with multiple editions of your program elements, code management becomes easy. VisualAge for Java keeps you on the right track.

Chapter 7. What else you can do

Introduction to what else you can do

You have already seen many of the interesting things that you can do in VisualAge for Java, but there is much more. This section gives you some more detail on the following features of VisualAge for Java:

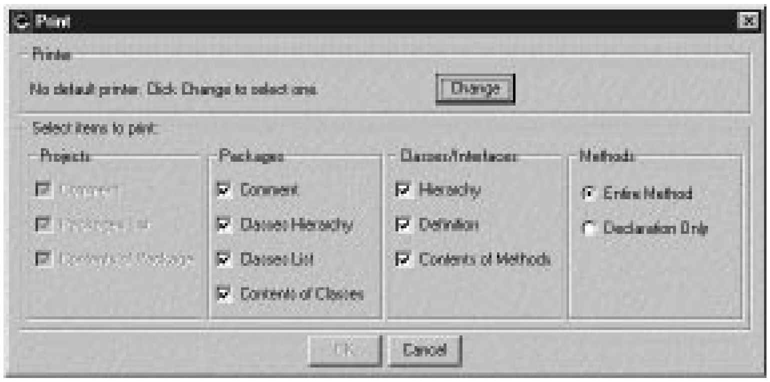
- “Printing program elements”
- “Navigating” on page 90
- “Searching” on page 94
- “Browsing” on page 97
- “Debugging” on page 100
- “Support for JavaBeans” on page 104
- “Customizing the Workspace” on page 112

Printing program elements

VisualAge for Java gives you several options for printing program elements. You can print projects, packages, classes, interfaces, or methods. When you print a program element that is composed of other program elements, you have the option of printing these other program elements. For example, when you print a package, you can also print the classes in the package.

To print a program element:

1. Select the program element and select **Print** from the **Selected** menu or from the pop-up menu for the program element. The Print secondary window appears.



No default printer. Click Change to select one.

Projects
☑ Comment
☑ Packages List
☑ Contents of Packages

Packages
☑ Comment
☑ Classes Hierarchy
☑ Classes List
☑ Contents of Classes

Classes/Interfaces
☑ Hierarchy
☑ Definition
☑ Contents of Methods

Methods
● Entire Method
○ Declaration Only

2. If no default printer has been selected, a message appears asking you to select one. To select a printer, select **Change** and select a printer from the Printer Selection secondary window.
3. The items that you can select to print depend on what kind of program element you are printing:
   - Selections under **Projects** are available if you are printing a project.
   - Selections under **Packages** are available if you are printing a project or a package.
   - Selections under **Classes/Interfaces** are available if you are printing a project, package, class, or interface.
   - Selections under **Methods** are always available.
4. By default, all the items under **Projects**, **Packages**, and **Classes/Interfaces** are selected, and **Entire Method** is selected under **Methods**. Change these selections if you want and select **OK** to start printing.

Changing the default printer

You can change the default printer by selecting **Change** in the Print secondary window or by selecting **Print Setup** from the **File** menu of any window.

Navigating

VisualAge for Java gives you many different ways to look at your code. This section gives you a brief overview of the primary windows in VisualAge for Java and tells you how to move from one window to another.

Moving between windows

Every window in VisualAge for Java has a **Window** menu. You can move between windows by selecting the window you want from this menu.

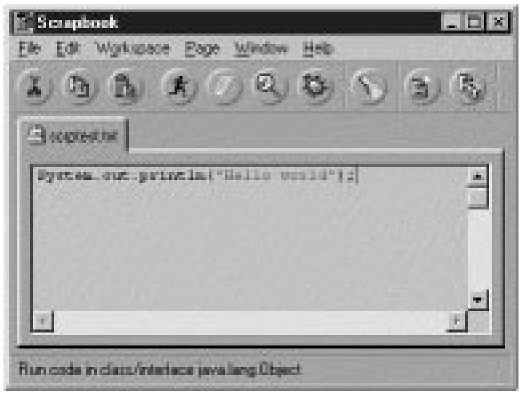
If the window you select is already open, it becomes the active window. If the window you want is not open, it is opened and becomes the active window. If you select **Switch To** in the **Window** menu, you can select from any of the windows that are currently open.

In addition to being opened explicitly by you, some windows are also opened by VisualAge for Java as you perform your development tasks. For example, suppose you run a program by selecting a class in the Workbench window and selecting **Run** from the **Selected** menu. If there is an active breakpoint in your program, the Debugger window opens when the breakpoint is reached. To return to the Workbench window, select **Workbench** from the **Window** menu in the Debugger window. See “Debugging” on page 100 for more details on breakpoints.

Windows you can open from the Window menu

Here is a summary of the windows that you can open from the **Window** menu:

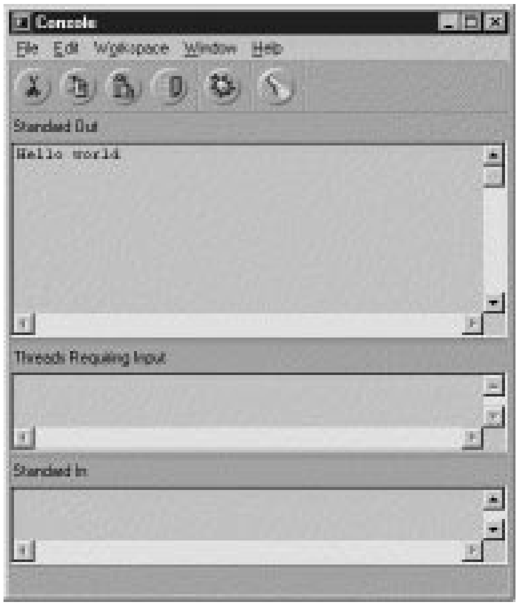
- Scrapbook - gives you a place to try out code. You can enter and run code fragments without making them a part of any package, project, or class.



```
System.out.println("Hello world");
```

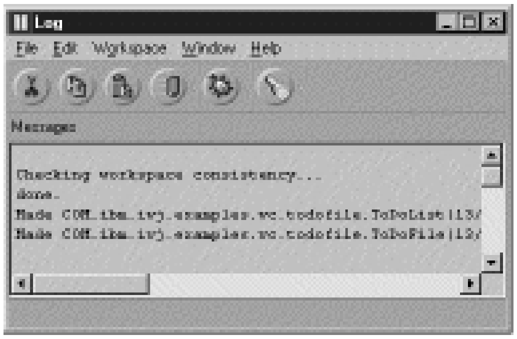
Run code in class/interface java.lang.Object

- Console - displays standard out. It also gives you an area for entering input to standard in. If more than one thread is waiting for input from standard in, you can select which thread gets the input.



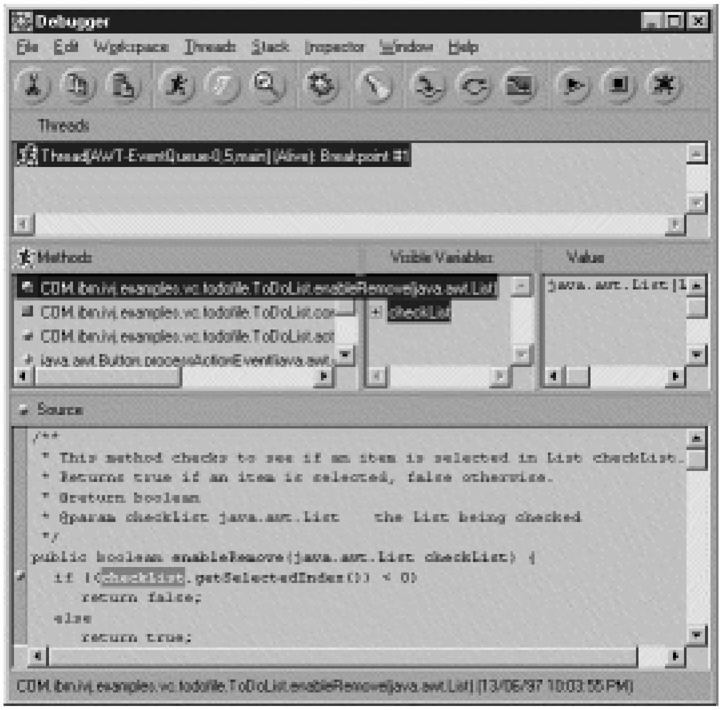
```
Hello world
```

- Log - displays messages and warnings from VisualAge for Java.



```
Checking workspace consistency...
done.
Made COM.ibm.ivj.examples.vc.todofile.ToDoList(13/
Made COM.ibm.ivj.examples.vc.todofile.ToDoFile(12/
```

- Debugger - displays running threads and the contents of their runtime stacks. In the Debugger you can also suspend and resume execution of threads. See “Debugging” on page 100 for more details.



```
Thread[AWT-EventQueue-0,5,main] (Alive): Breakpoint #1
```

```
COM.ibm.ivj.examples.vc.todofile.ToDoList.enableRemove(java.awt.List)
COM.ibm.ivj.examples.vc.todofile.ToDoList.con
COM.ibm.ivj.examples.vc.todofile.ToDoList.act
java.awt.Button.processActionEvent(java.awt.
```

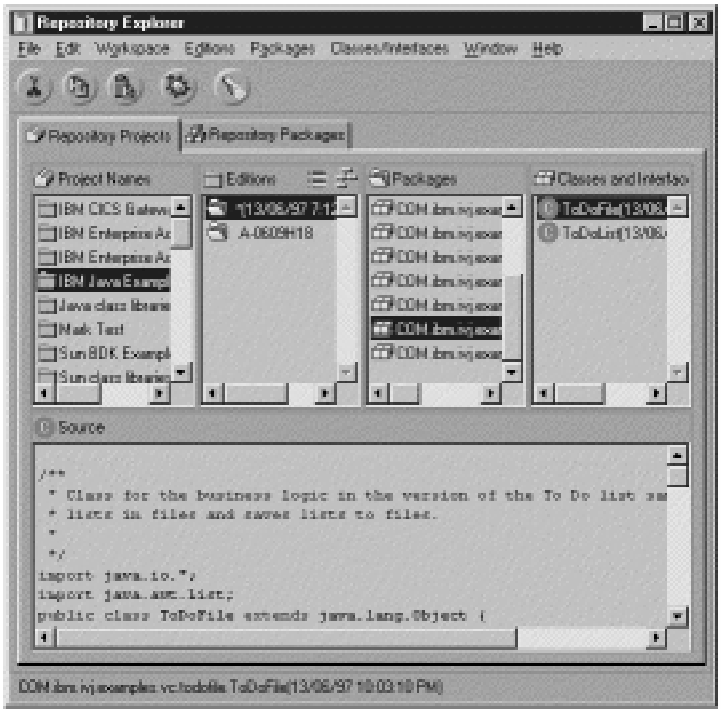
```
+ checkList
```

```
java.awt.List[1
```

```
/**
 * This method checks to see if an item is selected in List checkList.
 * Returns true if an item is selected, false otherwise.
 * @return boolean
 * @param checklist java.awt.List    the List being checked
 */
public boolean enableRemove(java.awt.List checkList) {
  if ((checkList.getSelectedIndex() < 0)
     return false;
  else
     return true;
```

```
COM.ibm.ivj.examples.vc.todofile.ToDoList.enableRemove(java.awt.List) (13/06/97 10:03:55 PM)
```

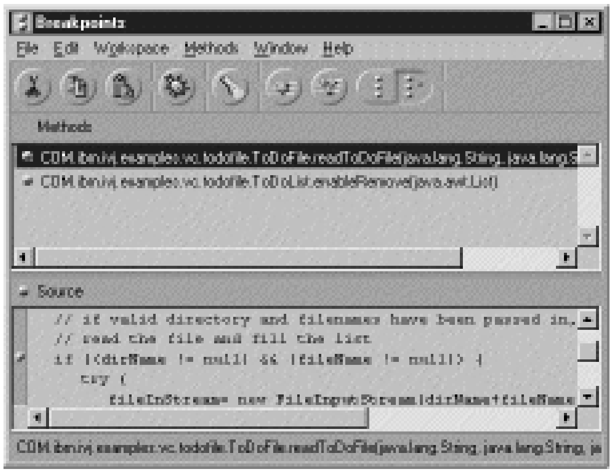
- Repository Explorer - displays all of the editions of program elements in the repository. See “Exploring the Repository” on page 85 for more details.



```
/**
 * Class for the business logic in the version of the To Do list sa
 * lists in files and saves lists to files.
 *
 */
import java.io.*;
import java.awt.List;
public class ToDoFile extends java.lang.Object {
```

COM.ibm.ivj.examples.vc.todofile.ToDoFile(13/06/97 10:03:10 PM)

- Breakpoints - displays all the breakpoints and allows you to manipulate them. You can enable or disable breakpoints. You can also clear selected breakpoints or all breakpoints.



COM.ibm.ivj.examples.vc.todofile.ToDoFile.readToDoFile(java.lang.String, java.lang.S

COM.ibm.ivj.examples.vc.todofile.ToDoList.enableRemove(java.awt.List)

```
   // if valid directory and filenames have been passed in,
   // read the file and fill the list
   if ((dirName != null) && (fileName != null)) {
      try {
         fileInStream= new FileInputStream(dirName+fileName
```

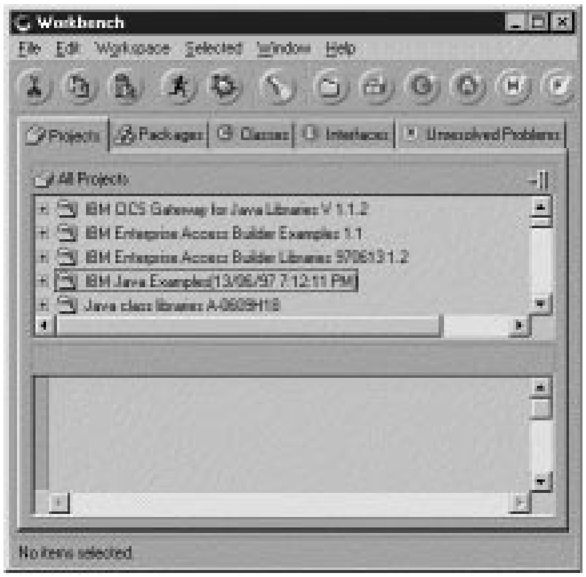
COM.ibm.ivj.examples.vc.todofile.ToDoFile.readToDoFile(java.lang.String, java.lang.String, ja

Searching

VisualAge for Java is designed to make it easy for you to find program elements and to move around within the interface. This section tells you how to take advantage of the search features of VisualAge for Java.

Searching for a program element

When you first start VisualAge for Java, the Workbench looks like this:



IBM CICS Gateway for Java Libraries V 1.1.2
IBM Enterprise Access Builder Examples 1.1
IBM Enterprise Access Builder Libraries 970613 1.2
IBM Java Examples(13/06/97 7:12:11 PM)
Java class libraries A-0609H18

No items selected.

VisualAge gives you several choices for searching for program elements. For example, in the Workbench, if you press a letter key, VisualAge for Java selects the first displayed program element that begins with that letter. If you press the same letter again, the next program element that begins with that letter is selected.

Now, suppose you want to look at the source for the String class in the standard Java class library. We'll go through two ways that you can search for the String class in VisualAge for Java.

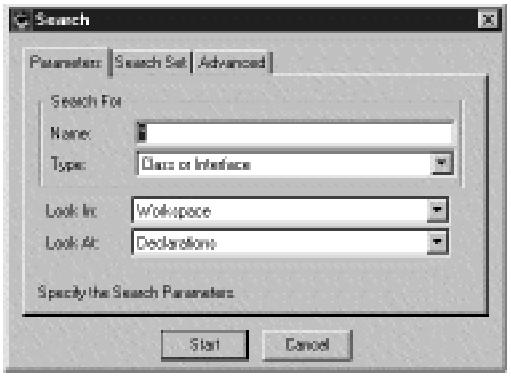
Searching with the Search secondary window

You can use the Search secondary window to perform powerful searches of the workspace. You can open the Search secondary window by doing any of the following:

- Select **Search** from the **Workspace** menu of any window.
- Select **Search** from the **Edit** menu of any window.
- Select a project or package and then select **Search inside** from the **Selected** menu. By default, this limits the search to the selected project or package.

- Select a class, interface, or method and then select **Search for References in** from the **Selected** menu. You can search for references in **Workspace**, **Project**, **Package**, or **Hierarchy**.
- Select a class, interface, or method and then select **Search for Declarations in** from the **Selected** menu. You can search for declarations in **Workspace**, **Project**, **Package**, or **Hierarchy**.
- Select **Search** in the tool bar of any window: . This is equivalent to selecting **Search** from the **Workspace** menu.

To keep things simple, suppose you select **Search** from the tool bar. The Search secondary window is displayed:

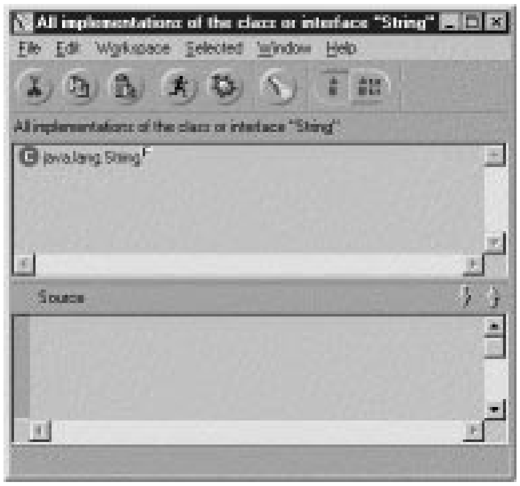


Name:
Type: Class or Interface
Look In: Workspace
Look At: Declarations
Specify the Search Parameters

The Search secondary window gives you many powerful options for scoping the range of your search. For example, you can search the entire workspace or just the subset of the workspace that you define on the **Search Set** page.

For now, to search for the String class:

1. Enter *String* in the **Name** field.
2. Select **Class or Interface** in the **Type** field. Select **Workspace** in the **Look In** field and **Declarations** in the **Look At** field. Select **Start** to begin the search. A list of all the matching program elements is displayed:


All implementations of the class or interface "String"
File Edit Workspace Selected Window Help
Source

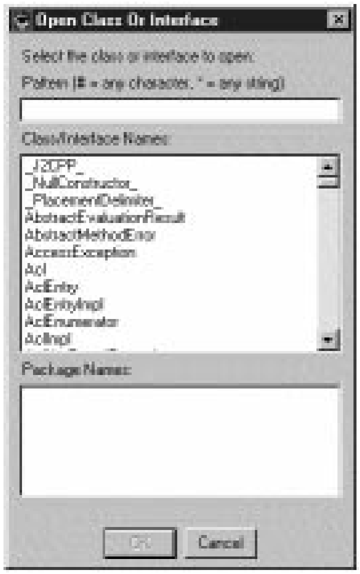
All implementations of the class or interface "String"

java.lang.String

When you select the String program element, the source for String is displayed in the Source pane. You can also select **Open** from the **Selected** menu to browse the String class.

Searching from the Workspace menu

You can also search for a program element by selecting one of the **Open** selections from the **Workspace** menu. For example, if you select **Open Class/Interface Browser** from the **Workspace** menu, the Open Class or Interface secondary window is displayed:



Select the class or interface to open:

Pattern (# = any character, * = any string)

Class/Interface Names:

J2CPP
NullConstructor
PlacementDelimiter
AbstractEvaluationResult
AbstractMethodError
AccessException
Acl
AclEntry
AclEntryImpl
AclEnumerator
AclImpl

Package Names:

As you enter *String* in the **Pattern** field, the **Class/Interface Names** list updates to show only the classes and interfaces that match what you have typed in so far. Select

String from the **Class/Interface Names** list and select **OK** to open a browser on the String class.

Browsing

VisualAge for Java gives you extensive facilities for browsing program elements.

In VisualAge for Java, you browse a program element by **opening** it. There are many ways to open a program element in VisualAge for Java, but for now here are two simple methods:

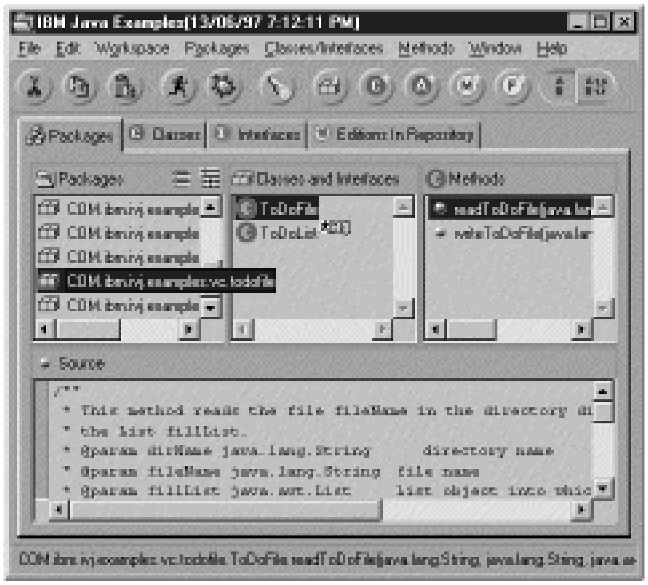
- Select the program element and select **Open** from the **Selected** menu or from the pop-up menu for the program element.
- Select the appropriate browser in the **Workspace** menu (**Open Class/Interface Browser**, **Open Package Browser**, or **Open Project Browser**) for the program element. A secondary window appears that lists all the classes and interfaces, packages, or projects in the workspace. Enter the name of your program element and select **OK**.

When you open a program element, a window appears that displays information about this program element. The following sections describe in more detail the windows that appear when you open each kind of program element.

Browsing a project

When you open a project, you get a window with four pages:

- The **Packages** page displays the hierarchy of packages contained in this project.
- The **Classes** page displays the hierarchy of classes contained in this project.
- The **Interfaces** page displays the interfaces contained in this project.
- The **Editions in Repository** page displays all the editions of this project.



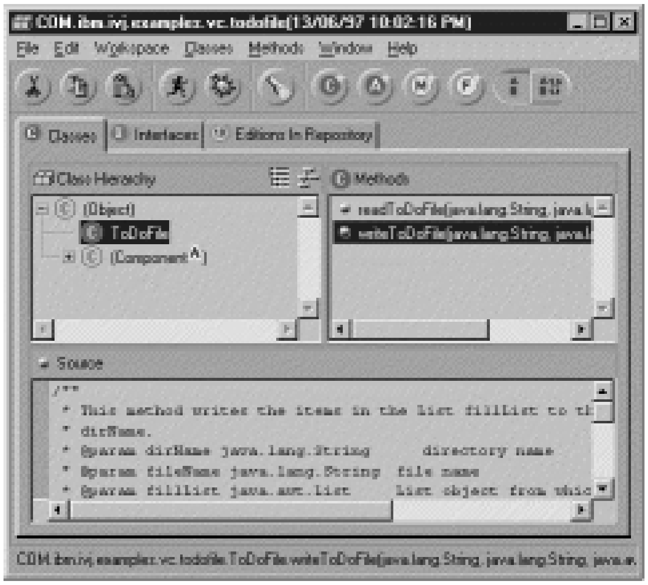
```
/**
 * This method reads the file fileName in the directory di
 * the list fillList.
 * @param dirName java.lang.String      directory name
 * @param fileName java.lang.String  file name
 * @param fillList java.awt.List     list object into whic
```

COM.ibm.ivj.examples.vc.todofile.ToDoFile.readToDoFile(java.lang.String, java.lang.String, java.aw

Browsing a package

When you open a package, you get a window with the three pages:

- The **Classes** page displays the hierarchy of classes contained in this package
- The **Interfaces** page displays the interfaces contained in this package
- The **Editions in Repository** page displays all the editions of this package


COM.ibm.ivj.examples.vc.todofile(13/06/97 10:02:16 PM)
File Edit Workspace Classes Methods Window Help
Classes
Interfaces
Editions In Repository
Class Hierarchy
Methods
(Object)
ToDoFile
(Component^A)
readToDoFile(java.lang.String, java.l
writeToDoFile(java.lang.String, java.l
Source

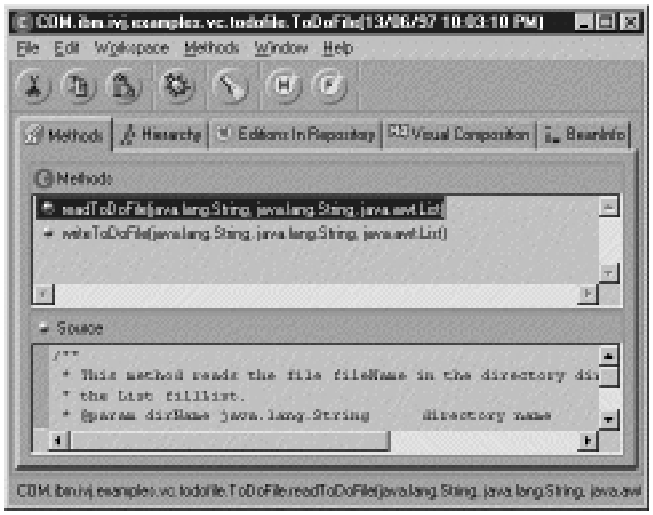
```
/**
 * This method writes the items in the list fillList to th
 * dirName.
 * @param dirName java.lang.String      directory name
 * @param fileName java.lang.String  file name
 * @param fillList java.awt.List     list object from whic
```

COM.ibm.ivj.examples.vc.todofile.ToDoFile.writeToDoFile(java.lang.String, java.lang.String, java.a

Browsing a class

When you open a class, you get a window with the five pages:

- The **Methods** page displays the methods contained in this class.
- The **Hierarchy** page displays the position of the class in the overall class hierarchy.
- The **Editions in Repository** page displays the editions of this class.
- The **Visual Composition** page displays the Visual Composition Editor. See “Using the Visual Composition Editor” on page 12 for more details.
- The **BeanInfo** page displays the JavaBean information for this class. See “The BeanInfo page” on page 105 for more details.



readToDoFile(java.lang.String, java.lang.String, java.awt.List)

writeToDoFile(java.lang.String, java.lang.String, java.awt.List)

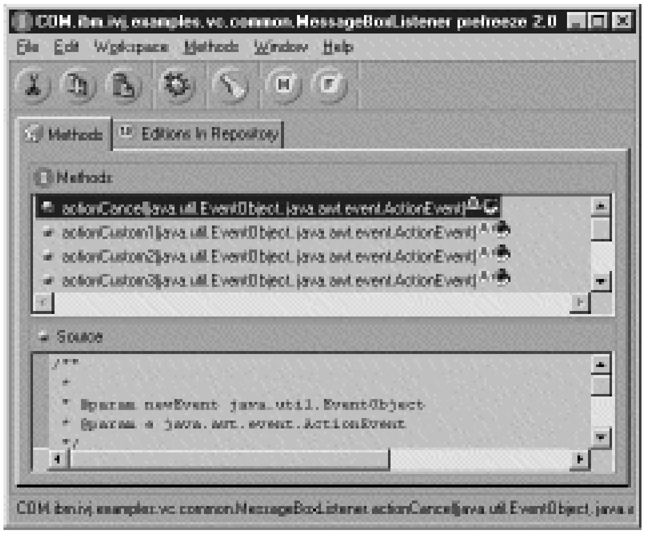
```
/**
 * This method reads the file fileName in the directory dir
 * the List filllist.
 * @param dirName java.lang.String      directory name
```

COM.ibm.ivj.examples.vc.todofile.ToDoFile.readToDoFile(java.lang.String, java.lang.String, java.awt

Browsing an interface

When you open an interface, you get a window with two pages:

- The **Methods** page displays the methods contained in this interface
- The **Editions in Repository** page displays the editions of this interface


COM.ibm.ivj.examples.vc.common.MessageBoxListener prefreeze 2.0
File Edit Workspace Methods Window Help
Methods
Editions In Repository
Methods
Source

actionCancel(java.util.EventObject, java.awt.event.ActionEvent)
actionCustom1(java.util.EventObject, java.awt.event.ActionEvent)
actionCustom2(java.util.EventObject, java.awt.event.ActionEvent)
actionCustom3(java.util.EventObject, java.awt.event.ActionEvent)

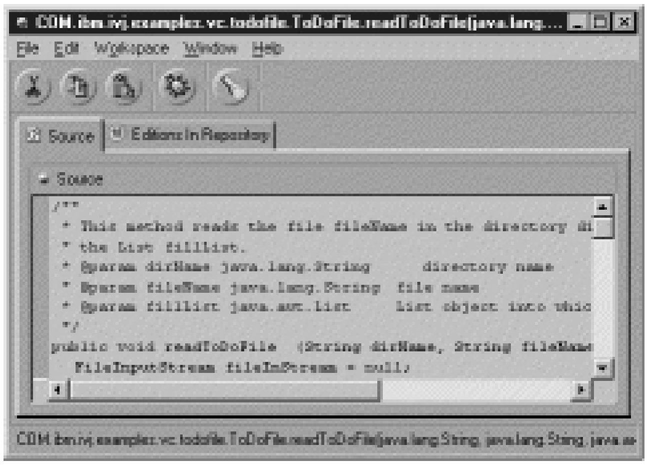
```
/**
 *
 * @param newEvent java.util.EventObject
 * @param e java.awt.event.ActionEvent
 */
```

COM.ibm.ivj.examples.vc.common.MessageBoxListener.actionCancel(java.util.EventObject, java.a

Browsing a method

When you open a method, you get a window with two pages:

- The **Source** page lists the source for the method
- The **Editions in Repository** page displays the editions of this method


COM.ibm.ivj.examples.vc.todofile.ToDoFile.readToDoFile(java.lang....
File Edit Workspace Window Help
Source
Editions In Repository
Source

```
/**
 * This method reads the file fileName in the directory di
 * the List filllist.
 * @param dirName java.lang.String      directory name
 * @param fileName java.lang.String  file name
 * @param filllist java.awt.List     List object into whic
 */
public void readToDoFile  (String dirName, String fileName
  FileInputStream fileInStream = null;
```

COM.ibm.ivj.examples.vc.todofile.ToDoFile.readToDoFile(java.lang.String, java.lang.String, java.aw

Debugging

VisualAge for Java includes a visual debugger with a rich set of features. This section outlines some of these features.

We begin by showing you how to set breakpoints. Then we discuss the features of the Debugger window and the Breakpoints window.

Setting breakpoints

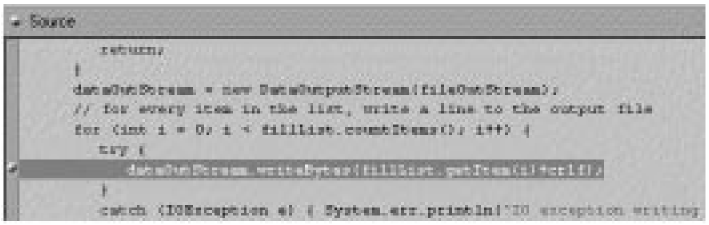
In VisualAge for Java, you can set breakpoints in any text pane that is displaying source. Suppose that you want to set a breakpoint in the *writeToDoFile* method in the ToDoFile class from the To-Do File program.

To set this breakpoint:

1. Display the source for the method.
    - Select the ToDoFile class in the Workbench. Expand the class to show its methods.
    - Select the *writeToDoFile* method. The source for the method is shown in the **Source** pane.
2. Double-click mouse button 1 in the left margin of the **Source** pane beside the following line (in the loop that writes items):

    ```
    dataOutStream.writeBytes(fillList.getItem(i)+crlf);
    ```

3. A breakpoint indicator appears in the margin of the **Source** pane beside this line:

    ```
        return;
      }
      dataOutStream = new DataOutputStream(fileOutStream);
      // for every item in the list, write a line to the output file
      for (int i = 0; i < fillList.countItems(); i++) {
        try {
          dataOutStream.writeBytes(fillList.getItem(i)+crlf);
        }
        catch (IOException e) { System.err.println("IO exception writing
    ```

You can also set a breakpoint on a line that does not already have a breakpoint by following these steps:

1. Move the cursor to the line.
2. Click mouse button 2 and select **Insert/Remove Breakpoint** from the pop-up menu.

Removing breakpoints

To remove a breakpoint, double-click mouse button 1 on the breakpoint indicator. You can also remove a breakpoint by following these steps:

1. Move the cursor to the line.
2. Click mouse button 2 and select **Insert/Remove Breakpoint** from the pop-up menu.

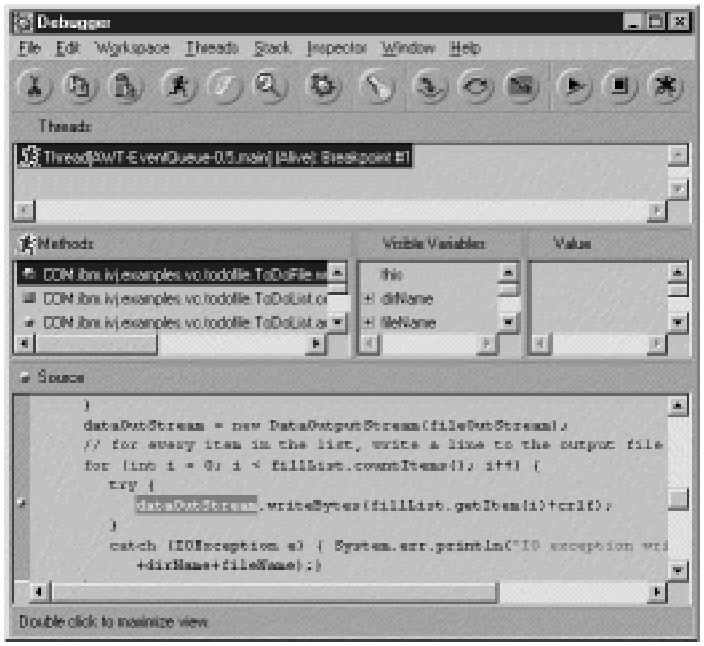
Try removing the breakpoint you just set. Now reset it. You will be using this breakpoint in the next section to examine the features of the Debugger window and the Breakpoints window.

Using the Debugger window and the Breakpoints window

You can open the Debugger window at any time by selecting Debugger from the Window menu. The Debugger window opens automatically when the program you are executing reaches an active breakpoint or has an unhandled exception.

Now that we have set a breakpoint, let's run the To-Do File program to see what happens:

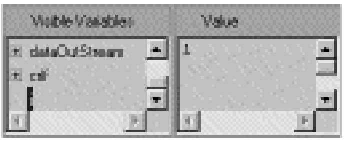
1. In the Workbench, select the ToDoList class. Select **Run** then **Run Main** from the **Selected** menu. Select **Run** in the Command Line Argument window.
2. When the To-Do File program appears, add some items to the **To-Do List** and then select **Save To-Do File**. When the Save To-Do File dialog appears, enter a file name and select **Save**. The Debugger window appears. It should look like this:


Debugger
File Edit Workspace Threads Stack Inspector Window Help
Threads
Thread[AWT-EventQueue-0,5,main] (Alive): Breakpoint #1
Methods
COM.ibm.ivj.examples.vc.todofile.ToDoFile.w
COM.ibm.ivj.examples.vc.todofile.ToDoList.o
COM.ibm.ivj.examples.vc.todofile.ToDoList.a
Visible Variables
this
dirName
fileName
Value
Source
Double click to maximize view.

```
    }
    dataOutStream = new DataOutputStream(fileOutStream);
    // for every item in the list, write a line to the output file
    for (int i = 0; i < fillList.countItems(); i++) {
       try {
          dataOutStream.writeBytes(fillList.getItem(i)+crlf);
       }
       catch (IOException e) { System.err.println("IO exception wri
          +dirName+fileName);}
```

The thread you are debugging is selected in the **Threads** list. In the **Methods** list, *writeToDoFile* (the method in which you set the breakpoint) is selected. The **Source** pane shows the source where the breakpoint is set.

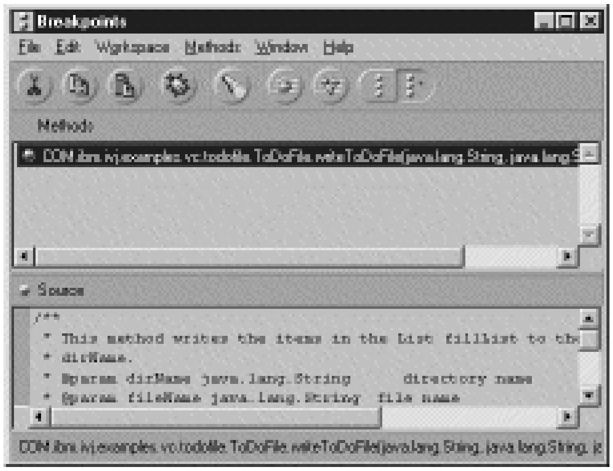
3. Select **Resume** from the tool bar: ▶ to continue execution of the program. Because this breakpoint is inside a loop, the Debugger window displays again immediately.
4. Examine some of the variables in the **Visible Variables** list. For example, to see the value of the loop counter variable *i*, select **i** from the **Visible Variables** list. Its value appears in the **Value** list:



This value of the loop counter is exactly what you would expect after the loop has been executed once.

5. Now let's disable this breakpoint:

   - Select **Breakpoints** from the **Window** menu. The Breakpoints window appears:

     COM.ibm.ivj.examples.vc.todofile.ToDoFile.writeToDoFile(java.lang.String, java.lang.S

     ```
     /**
      * This method writes the items in the List fillList to the
      * dirName.
      * @param dirName java.lang.String      directory name
      * @param fileName java.lang.String  file name
     ```

     COM.ibm.ivj.examples.vc.todofile.ToDoFile.writeToDoFile(java.lang.String, java.lang.String, ja

   - The Breakpoints window displays all the breakpoints that you have set in the workspace. The **Methods** pane lists all the methods in which you have set breakpoints. The **Source** pane displays the source for the method that is selected in the **Methods** pane.

   - To disable your breakpoints, select **Disable** from the tool bar: The breakpoint indicator changes colors to show that it is disabled.

   - Select **Debugger** from the **Window** menu to return to the Debugger window.

6. You can update and save code in the **Source** pane of the Debugger window. When you resume execution of the program, you see the changes you made to the code. For example, suppose that you wanted to change the *writeToDoFile* method so that items were written to the file in reverse order. You could make this change by modifying the beginning of the **for** loop to look like this:

   ```
   for (int i = fillList.countItems()-1; i >= 0; i--) {
   ```

   Make this change in the Source pane of the Debugger window, and then select **Save** from the **Edit** menu.

7. Now select **Resume** from the tool bar to continue execution of the program. In the To-Do File program, add the following values to the **To-Do List** and then select **Save To-Do File** to save them to a file:

- *item A*
- *item B*
- *item C*
- *last item*

8. Now select **Open To-Do File** and open the file you just saved. The **To-Do List** should look like this:

    - *last item*
    - *item C*
    - *item B*
    - *item A*

Before you continue, return to the Breakpoints window and enable your breakpoints again by selecting the **Enable** tool from the tool bar.

Support for JavaBeans

VisualAge for Java includes support for JavaBeans. This section gives you a very brief introduction to JavaBeans and some details on how VisualAge for Java supports them.

What are JavaBeans?

JavaBeans are Java objects that behave according to the JavaBeans specification. JavaBeans (or, more simply, **beans**) are reusable software components that you can visually connect together and manipulate in a development environment like VisualAge for Java. The method signatures and class definition of a bean follow a pattern that permits environments like VisualAge for Java to determine their properties and behavior. This ability for a beans-aware environment to determine the characteristics of a bean is called **introspection**.

Bean Features

Beans have three kinds of **features**:

- **Events**
- **Methods**
- **Properties**

You may remember seeing these three categories when you connected the beans of the To-Do File program in the Visual Composition Editor. A bean **exposes** a feature when it makes that feature available to other beans.

Here are brief descriptions of the three kinds of features:

1. **Events** are the events that the bean causes to occur. Other beans can register their interest in these events and be notified when they occur.
2. **Methods** are actions that a bean exposes for invocation by other beans. The bean methods are a subset of the public methods of the Java class that constitutes the bean.
3. **Properties** are the attributes exposed by a bean. Properties can be read and/or written. Properties can have the following characteristics:
   - A **bound** property triggers the *propertyChange* event when its value is changed.
   - A **constrained** property allows other beans to determine whether the value of the property can be changed.
   - An **indexed** property is an array.
   - A **hidden** property is not visible in the Visual Composition Editor.
   - An **expert** property should only be manipulated by expert users.
   - A **normal** property is one that is explicitly defined for the bean and that is neither hidden nor expert.

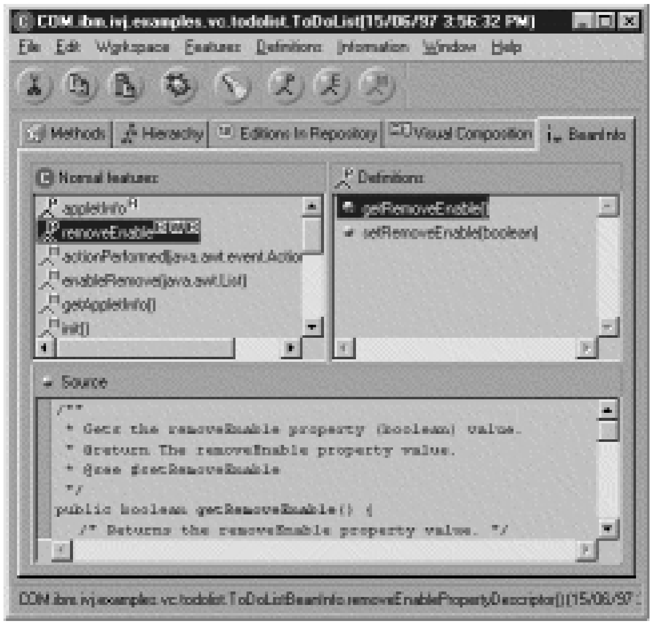
BeanInfo Classes

Beans can have accompanying BeanInfo classes. These classes explicitly describe the events, methods, and properties that a bean exposes. VisualAge for Java can generate BeanInfo classes for your beans. The BeanInfo class has the same name as the bean with the suffix "BeanInfo".

The BeanInfo class contains public methods that return information about the bean, including the class of the bean, the name of the class of the bean, and details about the events, methods, and properties of the bean.

The BeanInfo page

In VisualAge for Java you manipulate the characteristics of a bean in the BeanInfo page of the class browser.



COM.ibm.ivj.examples.vc.todolist.ToDoList(15/06/97 3:56:32 PM)
File Edit Workspace Features Definitions Information Window Help
Methods
Hierarchy
Editions In Repository
Visual Composition
BeanInfo
Normal features:
Definitions:
Source

appletInfo
removeEnable
actionPerformed(java.awt.event.Actio
enableRemove(java.awt.List)
getAppletInfo()
init()

getRemoveEnable()
setRemoveEnable(boolean)

```
/**
 * Gets the removeEnable property (boolean) value.
 * @return The removeEnable property value.
 * @see #setRemoveEnable
 */
public boolean getRemoveEnable() {
  /* Returns the removeEnable property value. */
```

COM.ibm.ivj.examples.vc.todolist.ToDoListBeanInfo.removeEnablePropertyDescriptor()(15/06/97

The top left pane lists the features of the bean. You can specify the kinds of features the BeanInfo page shows by selecting an entry under **Show** in the **Features** menu. The following groups of features are available:

- **All** All features in the bean, including features that were generated by VisualAge for Java
- **Normal** Features you explicitly defined for the bean
- **Property** Properties
- **Event** Events
- **Method** Methods
- **Hidden** Hidden features
- **Expert** Expert features

When you select a feature, VisualAge for Java lists information in the top right pane depending on what kind of feature is selected:

- **Event** Interface, listener methods, add listener method, remove listener method
- **Property** Type, read method, write method
- **Method** Signature

The top right pane lists the program elements that are associated with the selected feature. If you select one of the program elements, its source is displayed in the bottom pane.

If you do not select a program element in the upper right pane, the bottom pane lists the bean information for the selected feature, including its description, display name, and whether or not it is expert or hidden.

Using the BeanInfo Page

How would you use the BeanInfo page to create and manipulate the features of a bean? In this section we'll show you how you can modify your To-Do File program so that it uses a *mode* property that you create in the BeanInfo page. This property will control the mode of the file dialog: save or load.

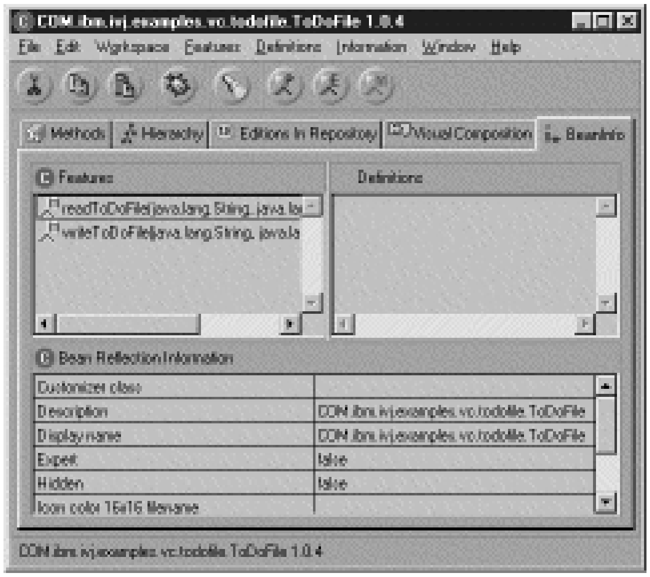
Here is a brief outline of what you will be doing:

1. Create a property called *mode* for the ToDoFile class in the BeanInfo page.
2. Create a new file dialog bean in the ToDoFile class to replace the two file dialog beans in the ToDoList class. Connect this file dialog to the new mode property.
3. Modify the *readToDoFile* and *writeToDoFile* methods in the ToDoFile class. Make the updated methods available as features.
4. Remove the old file dialog beans from the ToDoList class and reconnect the ToDoFile1 bean.

Before you start, ensure that you have versioned the latest edition of the package that contains your ToDoFile and ToDoList classes.

To create a new property in the ToDoFile class:

1. In the Workbench, select the ToDoFile class and select **Open To** then **BeanInfo** from the **Selected** menu.
2. The BeanInfo page appears. It should look like this:



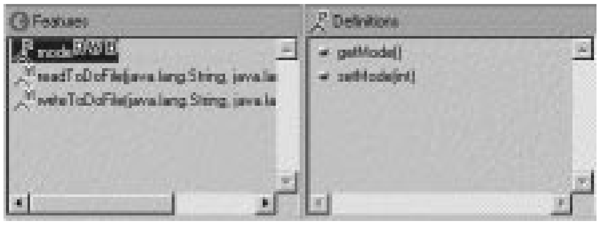
COM.ibm.ivj.examples.vc.todofile.ToDoFile 1.0.4
File Edit Workspace Features Definitions Information Window Help
Methods
Hierarchy
Editions In Repository
Visual Composition
BeanInfo
Features
Definitions
Bean Reflection Information

readToDoFile(java.lang.String, java.la
writeToDoFile(java.lang.String, java.la

Customizer class	
Description	COM.ibm.ivj.examples.vc.todofile.ToDoFile
Display name	COM.ibm.ivj.examples.vc.todofile.ToDoFile
Expert	false
Hidden	false
Icon color 16x16 filename	

COM.ibm.ivj.examples.vc.todofile.ToDoFile 1.0.4

3. Select **Create Property Feature** from the tool bar: . The New Property Feature SmartGuide appears.
4. In the New Property Feature SmartGuide, enter *mode* in the **Property name** field.
5. Select **int** in the **Property type** field.
6. Ensure that **Readable**, **Writeable**, and **bound** are selected. Select **Finish**. VisualAge for Java creates the new property, as well as functions to set and get it. A ToDoFileBeanInfo class is also created.
7. Now the **Features** and **Definitions** panes of the BeanInfo page should look like this:



mode (RWB)
readToDoFile(java.lang.String, java.la
writeToDoFile(java.lang.String, java.la

getMode()
setMode(int)

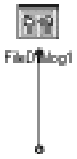
Now you are ready to add a file dialog bean to the ToDoFile class. This bean will replace the two file dialog beans in the ToDoList class, and it will get its mode from the *mode* property you just created.

To add a file dialog bean to the ToDoFile class and connect it to the *mode* property:

1. Select the **Visual Composition** page.
2. Select the **Containers** category in the beans palette:

3. Select the **FileDialog** bean:
4. Click mouse button 1 on the free-form surface. A new file dialog bean called FileDialog1 appears.
5. Select the new file dialog bean and double-click mouse button 1 to open the Properties window.
6. In the Properties window, enter *c:\\* as the **directory** value and close the Properties window.
7. Now make a connection to set the mode of the file dialog to be equal to the *mode* property you created:
   - Click mouse button 2 on the free-form surface and select **Connect** from the menu that appears. The Start connection from window appears.
   - In the Start connection from window, select **mode** from the **Property** list and select **OK**.
   - To complete the connection, click mouse button 1 on the file dialog bean and select **All Features** from the pop-up menu that appears. In the Connect property named window that appears, select **mode** from the **Property** list and select **OK**.
8. Select **Save Bean** from the **File** menu.

Now the free-form surface for the ToDoFile class should look like this:



Now you are ready to modify the *readToDoFile* method:

1. Select the **Methods** page and select **readToDoFile** from the **Methods** list. The source for this method appears in the Source pane.
2. The *readToDoFile* method will be getting the directory and file values directly from the file dialog, so you can remove the first two parameters in the method declaration. The modified signature should look like this:

   ```
   public void readToDoFile (List fillList)
   ```

3. Now, add statements at the very beginning of the method to set the *mode* property, to call *show()* and *dispose()* for the file dialog, and to get the file and directory values from the file dialog. After you have made all the changes, the beginning of the method should look like this:

```
public void readToDoFile (List fillList) {
  setMode(java.awt.FileDialog.LOAD);
  getFileDialog1().show();
  getFileDialog1().dispose();
  String fileName = getFileDialog1().getFile();
  String dirName = getFileDialog1().getDirectory();
  FileInputStream fileInStream = null;
  ...
```

4. Select **Save** from the **Edit** menu to save your changes.

Now make similar changes to the *writeToDoFile* method:

1. Select **writeToDoFile** from the **Methods** list in the **Methods** page.
2. Modify the beginning of the *writeToDoFile* method to look like this:

   ```
   public void writeToDoFile(List fillList) {
     setMode(java.awt.FileDialog.SAVE);
     getFileDialog1().show();
     getFileDialog1().dispose();
     String fileName = getFileDialog1().getFile();
     String dirName = getFileDialog1().getDirectory();
     FileOutputStream fileOutStream = null;
     ...
   ```
3. Select **Save** from the **Edit** menu to save your changes.

Now that you have updated these methods, make them available as features:

1. Select the BeanInfo page and select **Add Available Features** from the **Features** menu. The Add available features window appears.
2. In the Add available features window, select **readToDoFile(java.awt.List)** and **writeToDoFile(java.awt.List)** from the list and select **OK**.
3. Now both *readToDoFile* and *writeToDoFile* should appear in the **Features** pane of the BeanInfo page.

You have finished all your updates to the ToDoFile class. Now you are ready to update the ToDoList class by removing the old file dialog beans and revising the connections.

1. In the Workbench, select the ToDoList class. If you see an "x" beside the ToDoList class, it means that there is an unresolved problem. The connections between the **Open To-Do File** and **Save To-Do File** buttons and the ToDoFile1 bean are no longer valid because you have changed the number of parameters in the *readToDoFile* and *writeToDoFile* methods. Don't worry if this happens. You are just about to fix the problem.
2. Select **Open To** then **Visual Composition** from the **Selected** menu. The Visual Composition Editor opens on ToDoList.
3. Delete the following items by selecting them and pressing the Delete key. If a warning message appears about deleting the bean and all its connections, select

OK. If you make a mistake, remember that you can undo it by selecting **Undo** from the **Edit** menu:

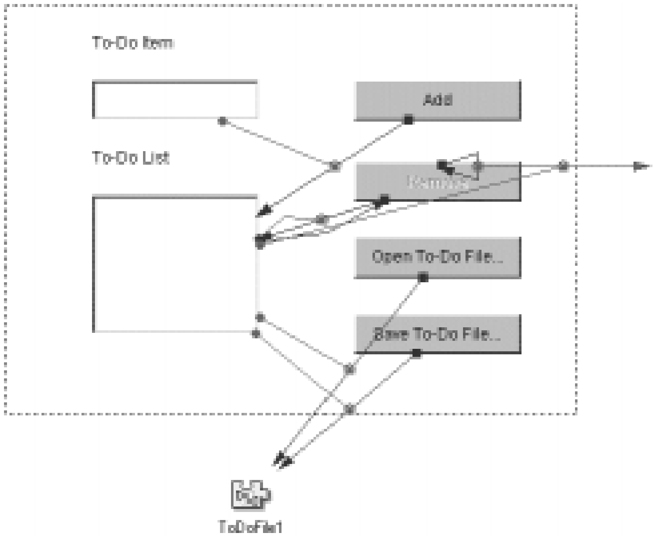
- The connection between the **Open To-Do File** button and the ToDoFile1 bean.
- The connection between the **Save To-Do File** button and the ToDoFile1 bean.
- The OpenFile bean.
- The SaveFile bean.

4. Create a new connection between the **Open To-Do File** button and the ToDoFile1 bean:

   - Select the **Open To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the menu that appears.
   - Click mouse button 1 on the ToDoFile1 bean and select **All Features** from the pop-up menu that appears.
   - Select **readToDoFile(java.awt.List)** from the **Method** list in the Connect event named window and select **OK**.
   - To complete this connection, select it and click mouse button 2. Select **Connect** then **fillList** from the pop-up menu that appears. Click mouse button 1 on the List and select **this** from the pop-up menu that appears.

5. Create a new connection between the **Save To-Do File** button and the ToDoFile1 bean:

   - Select the **Save To-Do File** button and click mouse button 2. Select **Connect** then **actionPerformed(java.awt.event.ActionEvent)** from the menu that appears.
   - Click mouse button 1 on the ToDoFile1 bean and select **All Features** from the pop-up menu that appears.
   - Select **writeToDoFile(java.awt.List)** from the **Method** list in the Connect event named window and select **OK**.
   - To complete this connection, select it and click mouse button 2. Select **Connect** then **fillList** from the pop-up menu that appears. Click mouse button 1 on the List and select **this** from the pop-up menu that appears. Now the free-form surface should look like this:



6. Select **Save Bean** from the **File** menu.
7. Now run your revised program by selecting the ToDoList class in the WorkBench and selecting **Run** then **Run main** from the **Selected** menu. Test to ensure that you can open and save files.

Congratulations! You have completed an alternate implementation of the To-Do File program that takes advantage of the features of the BeanInfo page.

Customizing the Workspace

VisualAge for Java gives you a range of characteristics that you can change to customize the workspace to suit your own needs and tastes. This section shows you how to set customization options and gives you a brief overview of the items that you can customize.

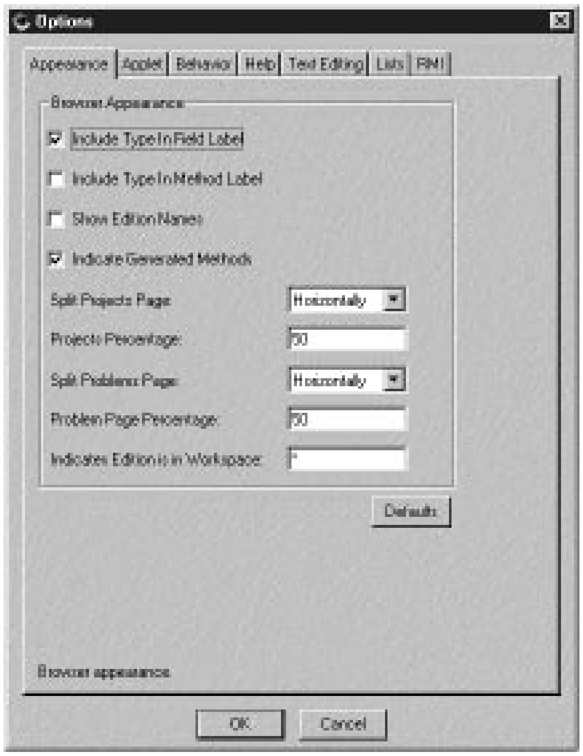
Setting customization options

In VisualAge for Java, you customize the workspace by setting options in the Options secondary window. Let's examine how this window works by setting the option that determines what happens when you double click on a program element.

By default, double clicking on a program element icon expands or collapses the program element tree beneath that icon. For example, when you double click on a package icon, VisualAge for Java displays all the classes that are in the package. You can change this behavior so that double clicking on a program element icon opens the program element in a browser of its own.

To specify that double clicking on a program element opens the program element:

1. Select **Options** from the **Workspace** menu. The **Options** secondary window appears.



Browser Appearance

☑ Include Type In Field Label

□ Include Type In Method Label

□ Show Edition Names

☑ Indicate Generated Methods

Split Projects Page: Horizontally

Projects Percentage: 50

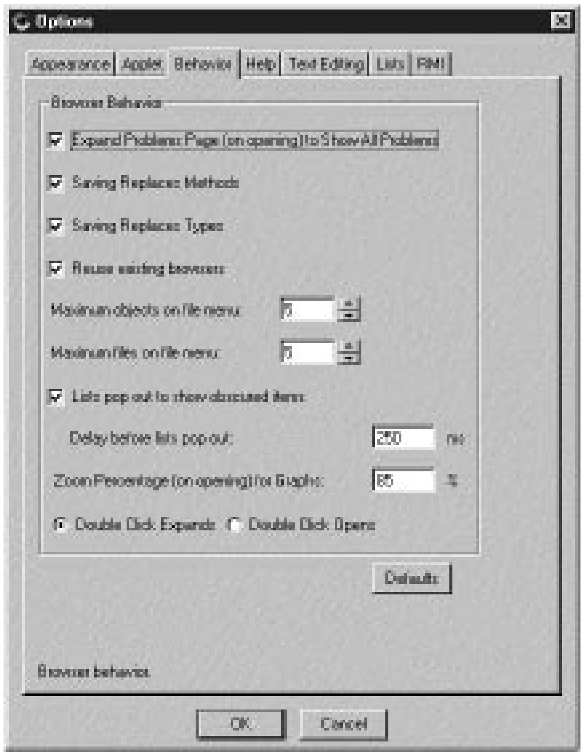
Split Problems Page: Horizontally

Problem Page Percentage: 50

Indicates Edition is in Workspace: *

Browser appearance

2. Select the **Behavior** page.



Browser Behavior

☑ Expand Problems Page (on opening) to Show All Problems

☑ Saving Replaces Methods

☑ Saving Replaces Types

☑ Reuse existing browsers

Maximum objects on file menu: 5

Maximum files on file menu: 5

☑ Lists pop out to show obscured items

Delay before lists pop out: 250 ms

Zoom Percentage (on opening) for Graphs: 85 %

● Double Click Expands ○ Double Click Opens

Browser behavior

3. In the **Behavior** page, select **Double Click Opens** and select **OK**.

Now, when you double click on a program element icon, the program element is opened in a browser.

To set all of the options on a page back to their default values, select **Defaults** and then select **OK**.

The following pages are available in the Options secondary window:

- The **Appearance** page displays options for specifying the appearance of the Workbench window and the other browsers, including:
  - Whether the type is included in field and method labels
  - Whether edition names are shown by default
  - Whether generated methods are marked
  - The proportion and orientation of text panes
  - The string that marks the edition (in a list of editions) that is currently in the workspace
- The **Applet** page displays options for specifying the width and height of the applet viewer window.

- The **Behavior** page displays options for specifying the behavior of the Workbench window and the other browsers, including:
  - Expansion of the **Unresolved Problems** page
  - Whether saving with a new name replaces methods and types
  - Whether existing browser windows are reused
  - The maximum number of objects and files on the **File** menu
  - Whether lists pop out to show obscured items and if so, the delay before the list pops out
  - The zoom percentage for graphs
  - Whether double clicking a program element expands it (displays the program elements included in it) or opens it.
- The **Help** page displays options for specifying which web browser is used to display the help information.
- The **Text Editing** page displays options for text editing panes, including:
  - Font settings for different text types, such as comments, keywords, and literals
  - Foreground and background colors
  - Style of indentation
- The **Lists** page displays options for specifying the font and colors of lists.
- The **RMI** page displays options for Remote Method Invocation (RMI).

Chapter 8. Accessing enterprise data

[ENTERPRISE]

You now have a solid grounding in how to use the core features of VisualAge for Java. You can create, debug, test, and manage editions of Java applets and applications. With VisualAge for Java Enterprise, there's still more under the hood for you to discover.

The VisualAge for Java Enterprise Access Builders let you easily code access to legacy applications and data. These builders generate JavaBeans for you, which you can connect to your user interfaces with the Visual Composition Editor.

Data Access Builder

Generates beans to access JDBC/ODBC-compliant relational databases. See “Data Access Builder” for details.

CICS Access Builder

Generates beans to access CICS transactions through the CICS Gateway for Java and CICS ECI client. See “CICS Access Builder” on page 127 for details.

RMI Access Builder

Generates proxy beans so you can distribute code for remote access, enabling Java-to-Java solutions. See “RMI Access Builder” on page 132 for details.

C++ Access Builder

Generates beans and C++ wrapper classes that let your Java programs access C++ DLLs. See “C++ Access Builder” on page 139 for details.

Note: Thusfar, Getting Started has led you through detailed, hands-on examples. Given the complexity of the material on the Enterprise Access Builders, we'll be taking a more conceptual approach here, with a special focus on understanding the deployment scenarios.

If you have purchased VisualAge for Java Professional and would like more details on upgrading to VisualAge for Java Enterprise, visit the VisualAge for Java Web site at `www.software.ibm.com/ad/vajava` or contact your IBM sales representative.

Data Access Builder

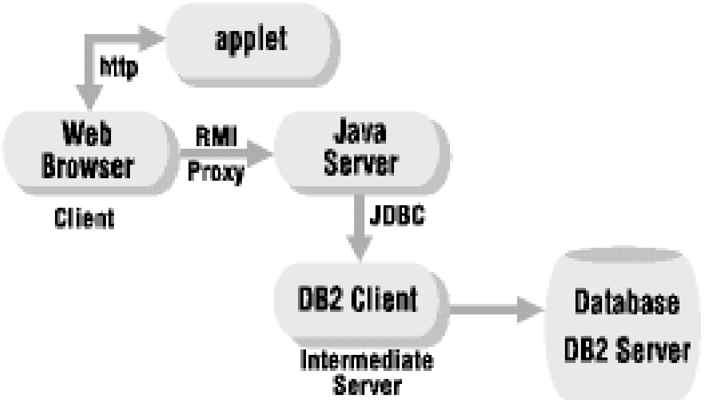
The VisualAge for Java Data Access Builder and other visual tools let you rapidly develop data access programs.

Before Java came along, you may have used CGI programs, HTML forms, and slower protocols to provide internet-based access to relational databases. The power and flexibility of Java deliver more robust access to databases.

JDK 1.1 includes the JDBC (Java Database Connectivity) API, which you can use to code your data access components. In much the same way that the ODBC standard defines an API that enables applications to access any ODBC-compliant database, the JDBC standard defines a common base on which other data access tools can be built. If you want to leverage existing ODBC databases using Java, Sun's JDBC-ODBC bridge driver translates JDBC calls into ODBC calls.

Thin-client model

There are various ways to deploy your data access program. The diagram below illustrates a 3-tiered thin-client model.



1. The end user downloads a thin Java client from an HTTP server to a Web browser. The applet only includes distributed front-end code, not the data access business logic.
2. The Web browser communicates with an intermediate server through remote method invocation calls using a TCP/IP protocol. Your generated data access classes reside on this server. Because of applet security restrictions, the applet and data access classes must reside on the same machine.
3. Your data access classes access the DB2 client through JDBC calls, which also reside on the intermediate server.
4. The DB2 client connects to the JDBC-compliant database.

The largest benefit of this scenario is that the end-user only requires a web browser to interact with the database. The DB2 client and all the data access Java code reside on an intermediate server.

VisualAge for Java makes developing this scenario easy. Use the Data Access Builder to generate your data access JavaBeans for you. The generated code includes not only code with the appropriate JDBC method calls, there is also a generated class that you may optionally use as a GUI prototype.

Now let's look at how this scenario is built. Here's a summary of the high-level steps:

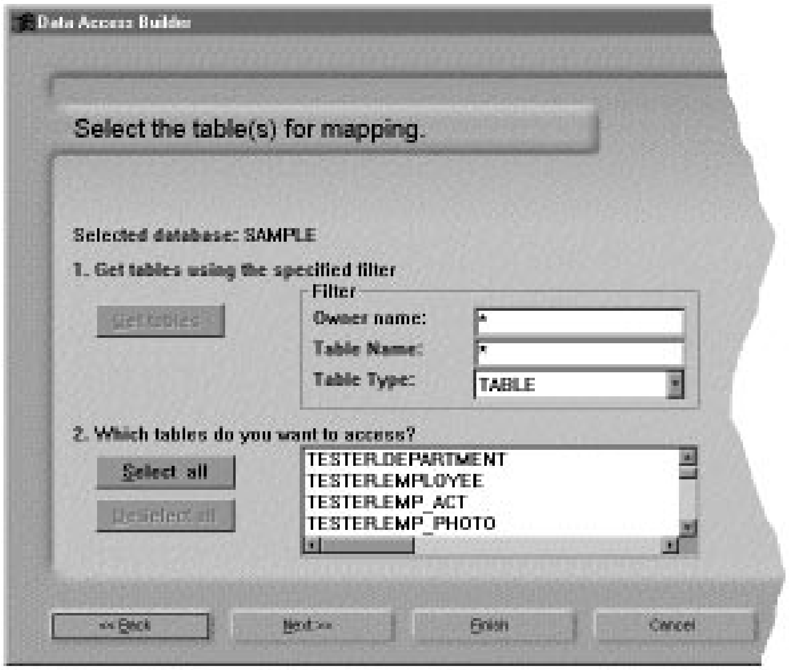
1. Generate the schema mapping
2. Customize the schema mapping
3. Generate the data access beans
4. Connect the generated beans to the user interface (this is already done for you if you use the generated GUI)
5. Distribute and deploy the GUI

Generating the schema mapping

The first step in creating your data access classes is to start the VisualAge for Java Data Access Builder:

- In the Workbench, select the package in which you want to place the access classes.
- From the pop-up menu, select **Tools** and then **Data Access** and **Create Data Access Beans**. The Data Access Builder now appears with a clear window.
- From the **File** pull-down, select **Map Schema** to create a new schema mapping. The Data Access Builder SmartGuide appears (shown below).

The Data Access Builder SmartGuide



Data Access Builder

Select the table(s) for mapping.

Selected database: SAMPLE

1. Get tables using the specified filter

Filter

Owner name: *

Table Name: *

Table Type: TABLE

2. Which tables do you want to access?

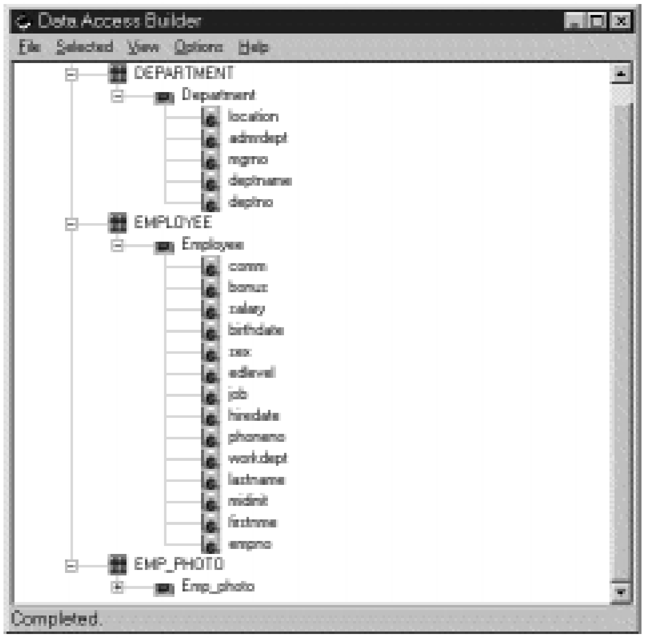
TESTER.DEPARTMENT
TESTER.EMPLOYEE
TESTER.EMP_ACT
TESTER.EMP_PHOTO

Use the Data Access Builder SmartGuide to create an initial schema mapping. This multi-page SmartGuide helps you specify the database tables that will be accessed by the generated Java classes. To define this information, you can filter through a list of tables and select to an existing table, or you can submit an SQL statement that dynamically joins the specified tables to create a schema. The database you select must be JDBC/ODBC-compliant so that the JDBC API calls that will form part of the generated Java code can access the database.

When you're done specifying input for the schema definition in the SmartGuide, select **Finish** to generate the schema mapping.

The generated schema mapping

The generated schema mapping now appears as a set of hierarchically-linked icons in the Data Access Builder window. The image below shows three schema mappings, for DEPARTMENT, EMPLOYEE, AND EMP_PHOTO.



The icons in the Data Access Builder window represent:

schema: representation of the database

mapping : defines characteristics of access to the database, and is used as input to generate the Java classes

attribute : corresponds to a table column

data ID: corresponds to a set of columns that uniquely identifies each row

This is the initial mapping. You can now select any mapping object and customize it.

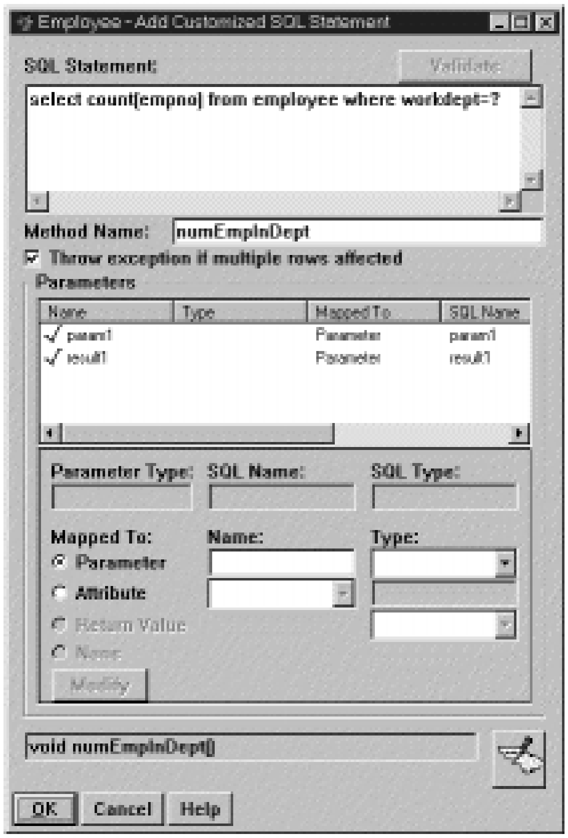
Customizing the schema mapping

Now that the SmartGuide has generated an initial schema mapping, you can modify it before generating the Java access methods. You can also add user-defined methods (for rows or collections of rows) either by specifying an SQL statement, or by accessing stored procedures defined in the database. Unused methods and attributes can be deleted from the initial schema mapping.

Customized SQL statements

To add or modify a method with an SQL statement:

- Select a mapping and from the pop-up menu, select **Methods**. (Select **Manager Methods** if the method returns a collection of rows.)
- Select **Add** to to bring up a secondary window to define the SQL statement.



SQL Statement:

```
select count(empno) from employee where workdept=?
```

Method Name: numEmpInDept

☑ Throw exception if multiple rows affected

Parameters

Name	Type	Mapped To	SQL Name
✓ param1		Parameter	param1
✓ result1		Parameter	result1

Mapped To: ● Parameter ○ Attribute

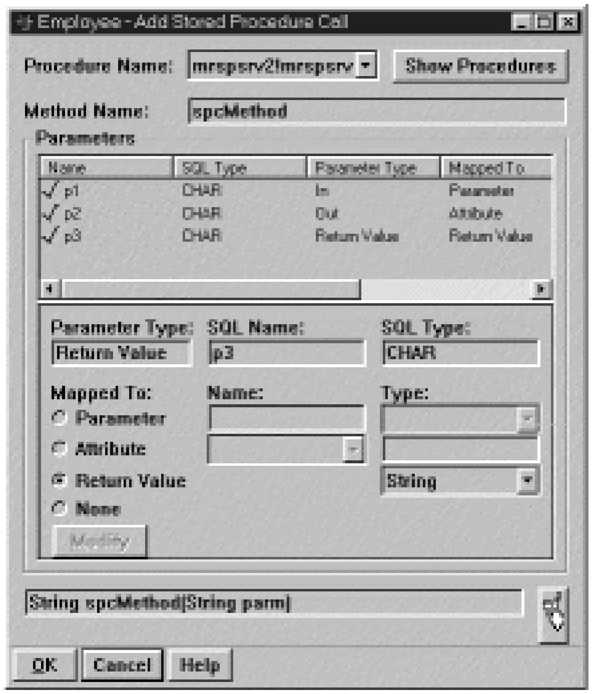
void numEmpInDept()

By entering an SQL statement in the **SQL Statement** field, you can define the parameters of a Java method. Click the **Validate** button to validate the SQL syntax and check that all database references are reflected in the schema. Once validated, the elements of your SQL statement are populated in the fields of the secondary window. You can then customize each parameter in terms of type, SQL name, attributes, and so on. This SQL statement is invoked on the DB2 client when the corresponding Java method is called.

Stored procedure calls

If your database contains stored procedures, you can define the parameters of Java methods that call the stored procedures.

- Select a schema and from the pop-up menu, select **Stored Procedures**. The Stored Procedures List secondary window appears.
- Select **Get Stored Procedures** to list all the stored procedures in the specified database. This database must be currently accessible.
- Select **Import** to retrieve the signature of the stored procedure call method.
- To define the stored procedure call method, select a mapping and from the pop-up menu, select **Methods** (or **Manager Methods** if the stored procedure returns a collection of rows). Select **Add** beneath the stored procedure call.



Procedure Name: mrspsrv2!mrspsrv

Method Name: spcMethod

Parameters

Name	SQL Type	Parameter Type	Mapped To
✓ p1	CHAR	In	Parameter
✓ p2	CHAR	Out	Attribute
✓ p3	CHAR	Return Value	Return Value

Parameter Type: Return Value
SQL Name: p3
SQL Type: CHAR

Mapped To:
○ Parameter
○ Attribute
● Return Value
○ None

Type: String

String spcMethod(String parm)

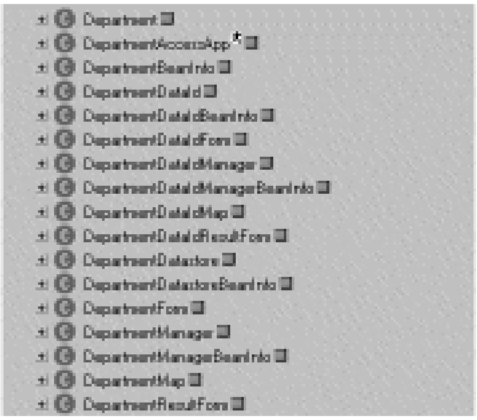
As with the method defined by an SQL statement, you can modify the attributes of existing stored procedure calls. You must ensure that you map the parameters correctly in this window. When called by your Java applet, the stored procedure runs on the DB2 server.

Generating the data access beans

From the Data Access Builder window, generating your data access Java classes is easy.

- From the **File** pull-down menu, select **Save**.

The data access classes are automatically generated and now appear in the package you had selected. Below is a sample of some of the generated classes.



Department
DepartmentAccessApp
DepartmentBeanInfo
DepartmentDataId
DepartmentDataIdBeanInfo
DepartmentDataIdForm
DepartmentDataIdManager
DepartmentDataIdManagerBeanInfo
DepartmentDataIdMap
DepartmentDataIdResultForm
DepartmentDatastore
DepartmentDatastoreBeanInfo
DepartmentForm
DepartmentManager
DepartmentManagerBeanInfo
DepartmentMap
DepartmentResultForm

Notice that a data access table icon is beside each class name, providing you with a visual reminder that these classes were generated by the Data Access Builder.

The classes have been generated and, by default, have been named using the mapping name. In the examples shown, classes have been generated using the *Department* mapping. The main types of generated code include:

- *DepartmentDatastore* represents and manages connections to the database.

- *Department* represents rows from the mapping. This class contains database access methods, including user-defined methods, if defined.

- *DepartmentDataId* represents the set of columns that uniquely identify a row. This class is only generated if the mapping specifies at least one data ID column.

- *DepartmentManager* enables you to select and work with a collection of rows. This class contains user-defined manager methods, if defined.

- *DepartmentDataIdManager* enables you to select and work with a collection of data IDs from the table. This class is only generated if the mapping specifies at least one data ID column.

- *DepartmentAccessApp* is an executable GUI that integrates the basic features of the mapping.

- BeanInfo classes enable the Visual Composition Editor to use the generated data access classes by providing notification of changes to object properties.

- Form support classes are prefabricated GUI components that reflect the basic features of your generated classes. You can add these beans in the Visual Composition Editor to help create the user interface that will access the data.

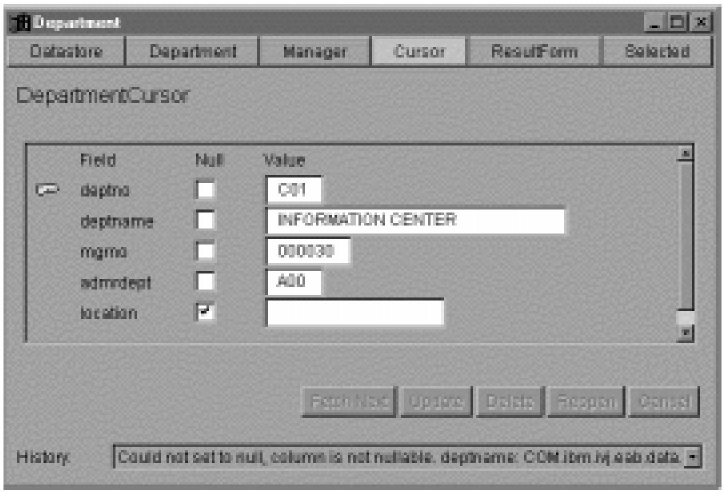
Because the Data Access Builder generates these classes, you no longer need to hand-code your JDBC calls.

Using the generated GUI

One of the classes generated by the Data Access Builder is a graphical user interface class. This class, called an AccessApp, includes a standardized layout that reflects the publicly accessible data and methods, as specified by the schema mapping. (In the list of generated classes seen earlier, *DepartmentAccessApp* is the AccessApp.) This simple GUI makes use of the generated classes to connect to datastores and manipulate data.

AccessApp classes help you with rapid application development. They're useful for demonstrating and prototyping database access, testing the generated classes, or using as your final program user interface. AccessApp classes can be invoked as applets or applications.

Below is one page of a generated AccessApp.



DepartmentCursor

	Field	Null	Value
🔑	deptno	□	C01
	deptname	□	INFORMATION CENTER
	mgrno	□	000030
	admrdept	□	A00
	location	☑	

History: Could not set to null, column is not nullable. deptname: COM.ibm.ivj.eab.data.

The AccessApp class lets you connect and disconnect from the database, and retrieve, update, and delete rows. Visual indicators also help you sift through the data. For example, the key icon in the screen above indicates that the **deptno** field represents a primary key column in the table.

With AccessApp classes, you can completely bypass programming the user interface in the creation of your data access program.

Connecting to your own GUI

Instead of using the AccessApp class, you can also create your own user interface. The Visual Composition Editor provides rich support for creating a data access user interface.

As you would create other user interfaces in the Visual Composition Editor, begin by adding buttons, labels, and other visual beans. Then add the generated nonvisual data access beans.

Adding visual data access beans to the free-form surface

1. From the left column of the bean palette, select the Enterprise Access category to display the Enterprise Access beans in the bean column. Several of these beans are intended for use in data access programs.
2. Select a data access bean from the bean column and add it to the free-form surface. Some examples include:

   - JDBC Datastore
   - Data Exception

3. You can also add one of the generated form support classes:

   a. In the Visual Composition Editor, select **Add Bean** from the **Options** pull-down menu.

   b. In the Add Bean dialog, click on **Browse** beside the **Class Name** field.

   c. In the Choose a valid class secondary window, select one of the generated form support classes and select **OK**.

   For example, *DepartmentForm* is a panel with entry fields for use with a single object, and *DepartmentResultForm* is a multicolumn list box tailored to reflect the class result set.

Adding nonvisual data access beans to the free-form surface

1. In the Visual Composition Editor, select **Add Bean** from the **Options** pull-down menu.
2. In the Add Bean dialog, click on **Browse** beside the **Class Name** field.
3. In the Choose a valid class secondary window, select one of the generated classes that contains business logic and select **OK**.
4. On the free-form surface, move the cursor to an area outside the visual section and click mouse button 1. The bean is now visually represented.

You need to repeat these steps for all the beans you want to import into the Visual Composition Editor.

To enable interface access to the methods and fields of the nonvisual data access beans, use the connection features of the Visual Composition Editor, as demonstrated in previous sections.

Distributing and deploying the GUI

The thin-client scenario illustrated at the outset of this section indicated that the data access client is distributed between the client applet and the intermediate server. To distribute the class representing the GUI, whether this is the AccessApp class (such as *DepartmentAccessApp* seen earlier) or your own GUI class, you can use the VisualAge for Java RMI Access Builder. This tool takes your bean and generates client-side and server-side proxies that communicate using RMI, as defined by the JDK 1.1 API.

For more information on distributing beans in a Java-to-Java, client-server environment, see “RMI Access Builder” on page 132.

You can now deploy the program by linking the distributed GUI class into a Web page and placing the other data access and RMI classes on the intermediate server.

You've done it! Putting together a web-based application to access a relational database is easy with VisualAge for Java. The Data Access Builder helps you define a schema mapping and generates Java code for you. This rapid application development environment also generates an AccessApp for you, so you can quickly test your code in a standardized interface.

CICS Access Builder

With the VisualAge for Java CICS Access Builder, accessing CICS transactions from your Java programs is made easy.

Before Java came along, you accessed your CICS transactions through the CICS Client using ECI (External Call Interface) API calls in your C or C++ programs. Then the Java wave brought with it the IBM CICS Gateway for Java, a Java version of the CICS client, which allows Java clients to access CICS transactions. The Java ECI method calls handled by the CICS Gateway for Java generally match the corresponding ECI method calls one-for-one.

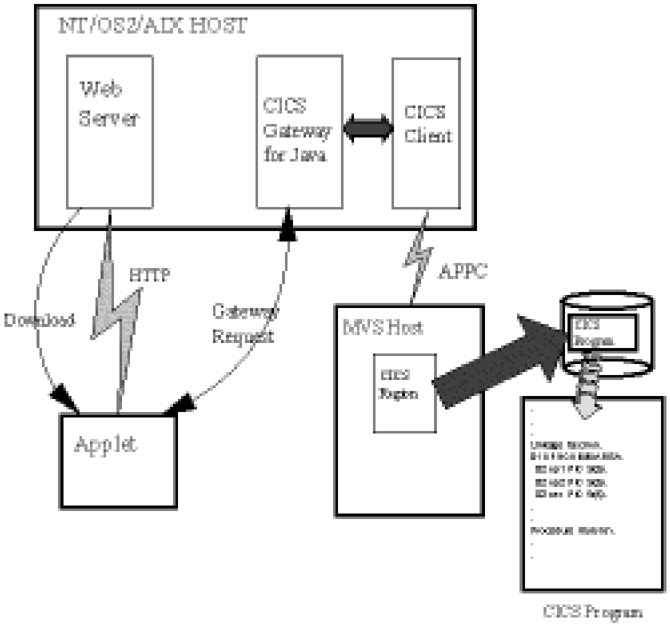
The common stumbling block introduced by both the CICS ECI and the Java ECI APIs, however, is that the data types in the calling programs (Java, C, C++) don't map well to the data types used in the CICS program, which is written in COBOL. Making the two programs communicate correctly to one another requires you to write code to perform these mappings. But no more! The CICS Access Builder creates a robust CICS access bean and associated classes for you; you no longer need to code the conversions between languages.

In this section, we'll review the high-level steps required to create Java-based, CICS access programs with VisualAge for Java:

- Use the Create COMMAREA Bean SmartGuide to generate a communications area bean
- Create an instance of the Unit of Work bean
- Connect the CICS access beans to the Java client user interface

CICS access overview

The following diagram illustrates the traditional path taken between your Java client and the CICS server.



The flow of requests are:

1. The end-user downloads the CICS access applet from an HTTP web server.
2. The Java client sends Java ECI requests and transfers data for the COBOL communications area to the CICS Gateway for Java.
3. The CICS Gateway for Java forwards the Java ECI requests as ECI requests to the CICS Client. Due to applet security restrictions, both the CICS Gateway for Java and the CICS Client reside on the same host as the web server.
4. The CICS Client forwards information to the MVS CICS server, where the CICS transactions reside.

By delivering the beans you need for step 2 above, VisualAge for Java makes CICS access programming a safer journey. The problems associated with data type conversions are handled for you.

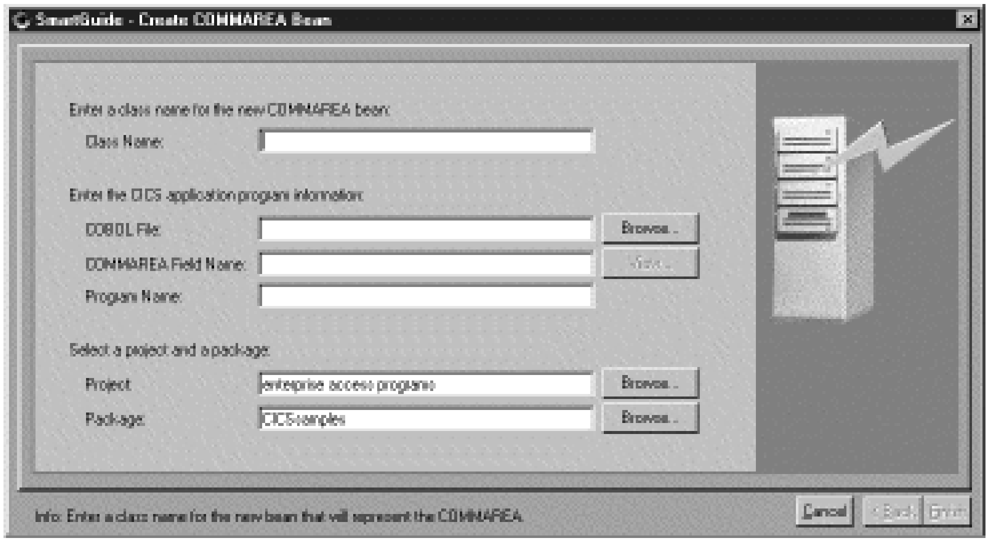
- The CICS Access Builder generates a communications area bean and other support classes.
- You can easily create an instance of the CICS Gateway for Java wrapper class, the Unit of Work bean, in the Visual Composition Editor.

Let's now see how these beans are created.

Using the Create COMMAREA Bean SmartGuide

The Create COMMAREA Bean SmartGuide leads you through the process of generating the communications area bean. To invoke the SmartGuide:

- In the Workbench, select a package into which you want to place the CICS COMMAREA bean and associated classes.
- From the pop-up menu, select **Tools**, then select **Host CICS Access** and **Create COMMAREA Bean**. The SmartGuide now appears.



SmartGuide - Create COMMAREA Bean

Enter a class name for the new COMMAREA bean:

Class Name:

Enter the CICS application program information:

COBOL File:

COMMAREA Field Name:

Program Name:

Select a project and a package:

Project: enterprise access programs

Package: CICSsamples

Info: Enter a class name for the new bean that will represent the COMMAREA.

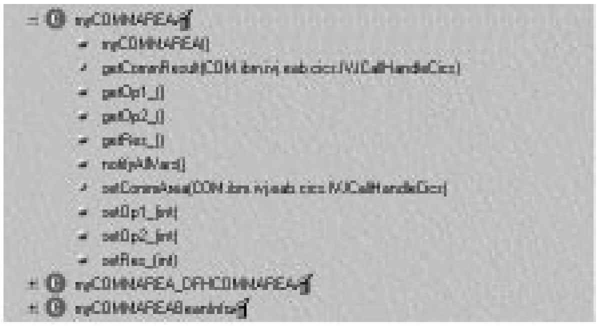
- In the **Class Name** field, enter a name for the new communications area (COMMAREA) bean.
- In the **COBOL File** field, enter the name of the local copy of the COBOL file.
- In the **COMMAREA Field Name** field, enter the record name of the communications area defined in the COBOL file. Often, the record name is DFHCOMMAREA.
- In the **Program Name** field, enter the name of the CICS program as it's known on the MVS host.

- Select **Finish**.

As an example, assume the following COBOL file was used as input:

```
identification division.
program-id. ADDER.
environment division.
data division.
working-storage section.
01 tmp pic a(40).
LINKAGE SECTION.
01  DFHCOMMAREA.
    02  op1  PIC S99999 DISPLAY.
    02  op2  PIC S99999 DISPLAY.
    02  res  PIC S99999 DISPLAY.
procedure division.
start-para.
    add op1 to op2 giving res.
    move 'ADDER transaction executed.' to tmp.
    EXEC CICS WRITE OPERATOR TEXT(tmp) TEXTLENGTH(27)
    ACTION(2) END-EXEC.
    EXEC CICS RETURN
    END-EXEC.
```

The SmartGuide generates three classes, each with a CICS symbol beside it.



```
- myCOMMAREA
    myCOMMAREA()
    getCommResult(COM.ibm.ivj.eab.cics.IVJCallHandleCics)
    getOp1_()
    getOp2_()
    getRes_()
    notifyAllVars()
    setCommArea(COM.ibm.ivj.eab.cics.IVJCallHandleCics)
    setOp1_(int)
    setOp2_(int)
    setRes_(int)
+ myCOMMAREA_DFHCOMMAREA
+ myCOMMAREABeanInfo
```

- *myCOMMAREA* is the main communication area class to invoke CICS transactions, and contains set and get methods for the basic data types of the CICS communications area and event triggering to notify listeners about changed properties.

- *myCOMMAREA_DFHCOMMAREA* contains all primitive type fields of one level of the CICS transaction communications area. Methods of this class are responsible for converting data to and from wire representation.

- *myCOMMAREABeanInfo* describes the bean features (events, methods, properties) of *myCOMMAREA* bean.

The Unit of Work bean

The workspace includes a class called IVJCicsUOWInterface that you can use as an interface to IBM CICS Gateway for Java in any CICS access program. This class acts as a wrapper that manages all requests between the Java client and the CICS Gateway for Java. As the focal point of all work on the Java client, it has been dubbed the Unit of Work bean.

Portions of this class are shown below:

```
public class UOWInterface {
      public void startUOW() {};
      public void endUOW() {};
      public void commitUOW() {};
      public void backoutUOW() {};
      public void invokeTxn(CommAreaBase bean) {};
      public void asynchInvokeTxn(CommAreaBase bean) {};
}
```

With the Unit of Work bean, you can:

- Start and end a unit of work with the CICS Gateway for Java
- Commit or roll back changes introduced during this unit of work
- Pass the communications area bean (generated by the CICS Access Builder) to the host-based CICS program, to run the transaction both synchronously and asynchronously.

You can create and use an instance of the Unit of Work bean in the Visual Composition Editor.

Connecting the CICS access beans to the user interface

You can connect the CICS access beans to the user interface in the Visual Composition Editor.

Follow these general steps to assemble the client:

- In the Workbench, select the class that represents the user interface. From the pop-up menu, select **Open To** and then **Visual Composition**.
- From the bean category of the bean palette, select the Enterprise Access category. Select the CICS Unit of Work bean, move the cursor to a nonvisual bean area and click mouse button 1. This creates a nonvisual part that represents an instance of the Unit of Work bean.

  CICS Unit of Work bean

- To import the communications area bean, select **Add Bean** from the **Options** pull-down menu.
- In the Add Bean dialog, click on **Browse** beside the **Class Name** field. In the Choose a valid class secondary window, select the communications area bean and select **OK**.
- For the connections, actions that interact with CICS transactions (for example, clicking a button) are represented by connecting the event-generating object (for example, the button) directly to the Unit of Work bean, which receives input from the communications area bean.

With the CICS Access Builder, it's that simple. Your host-based CICS transactions can be accessed by any user across the web!

RMI Access Builder

Are you trying to develop Java-to-Java solutions in a distributed, client-server environment? If so, chances are that you're facing one or both of the following problems:

- You find it difficult to integrate distribution-specific code with non-distributed, application-specific beans.
- If you've been coding with the RMI API supplied with JDK 1.1, you've noticed that the object distribution protocol doesn't support the distribution of bean events.

The VisualAge for Java RMI Access Builder helps you overcome these problems, enabling you to easily provide remote access to your beans in a client-server environment. In addition, the RMI Access Builder integrates your RMI development more tightly and generates source code for you that supports distribution over RMI:

- The RMI Access Builder generates server proxies, one of which you can use in the client program to access and act as the server bean.
- VisualAge for Java tightly integrates the RMI tool into the Workbench, enabling you to produce an end-to-end distributed solution from within a single development environment. (The RMI compiler bundled with JDK is run from the command-line, and isn't integrated into a development environment.)

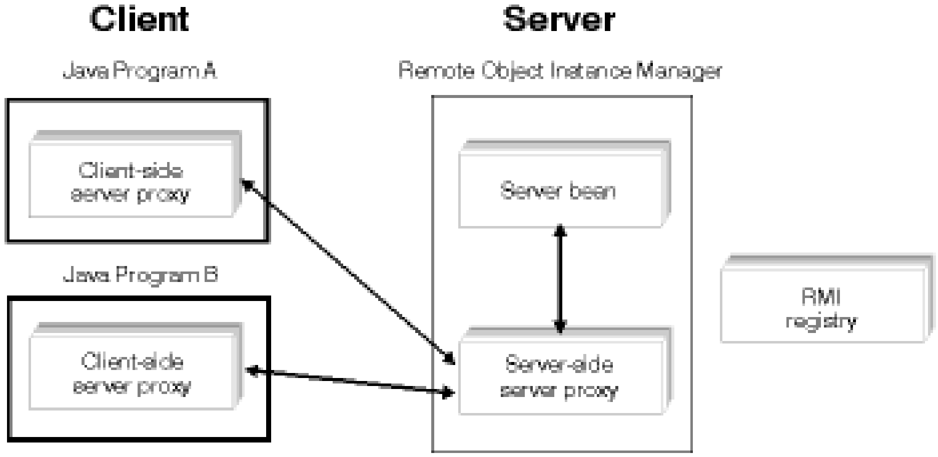
The RMI Access Builder provides a quick way to turn a local bean into a server bean by distributing it over RMI.

In this section, we'll review how to use these components to create Java-to-Java solutions.

- Use the Create Proxy Bean SmartGuide
- Create RMI stubs and skeleton classes
- Connect the client-side server proxy to your Java client
- Start the server programs

Overview of RMI access through proxy beans

The following diagram illustrates a typical RMI scenario:



server bean The bean that you would like to distribute and access remotely. Server bean methods can be invoked by your Java client program, and the server bean can generate events that are received by the client.

client-side server proxy A local representative of the remote server bean. The remote method access and event generation capabilities of the client-side server proxy allow you to treat the proxy as if it were the server bean itself. Because this proxy performs RMI initialization and the actual remote method invocation, the other code in your program does not need to deal with RMI code.

This local proxy can be connected to your Java client program as a non-visual bean in the Visual Composition Editor, and becomes part of your applet or application.

Multiple copies of clients from different systems or machines can access the same server-side server proxy, each in their own instance process.

server-side server proxy A companion class to the client-side server proxy, which facilitates the communication of the client-side server proxy over RMI. The server-side server proxy is deployed on the server to access the server bean, and to relay events and exceptions from the server bean back to the client-side server proxy. In effect, server events are recreated on the client.

Remote Object Instance Manager A separate, long-running server process that creates and monitors instances of the server bean through its associated server-side server proxy. The Remote Object Instance Manager can manage multiple server bean objects.

You provide the program bean that acts as the server bean. The rest of the code required to implement a distributed solution is generated by VisualAge for Java.

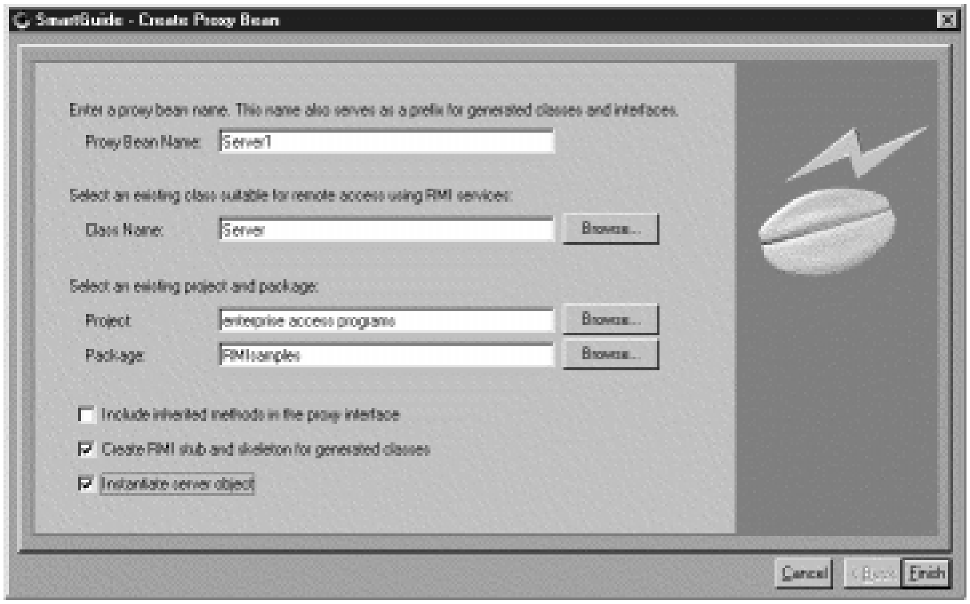
The steps to create a distributed solution are:

1. Run the RMI Access Builder against your server bean to create a proxy bean and associated classes and interfaces.
2. Generate RMI stubs and skeleton classes from the generated proxy bean, classes, and interfaces.
3. Connect the client-side server proxy to the user interface in the Visual Composition Editor.
4. Deploy your code and start the RMI registry and the Remote Object Instance Manager on the server.

Using the Create Proxy Bean SmartGuide

To begin the process of distributing the server bean, you first need to create a proxy bean:

- In the Workbench, select a class that you want to distribute over RMI. This will become the server bean. The generated classes and interfaces that make up the proxy bean will be placed in the same package that contains the class to be distributed.
- From the pop-up menu, select **Tools**, then select **Remote Bean Access** and **Create Proxy Beans**. The SmartGuide now appears.



SmartGuide - Create Proxy Bean

Enter a proxy bean name. This name also serves as a prefix for generated classes and interfaces.

Proxy Bean Name: Server1

Select an existing class suitable for remote access using RMI services:

Class Name: Server

Select an existing project and package:

Project: enterprise access programs

Package: RMIsamples

☐ Include inherited methods in the proxy interface

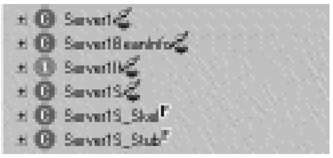
☑ Create RMI stub and skeleton for generated classes

☑ Instantiate server object

- In the **Proxy Bean Name** field, specify the name of the proxy bean. This name is assigned to the client-side server proxy, and serves as a prefix in the names of the other generated classes and interfaces.
- In the **Class Name** field, the server bean name is already filled in.

- Select the **Include inherited methods in the proxy interface** checkbox if you want the client to have access to the inherited methods of the server bean.
- Select the **Create RMI stub and skeleton for generated classes** checkbox if you want the SmartGuide to produce RMI stubs and skeletons automatically, using the generated classes and interfaces of the proxy bean as input. A stub/skeleton pair of classes is generated for the generated server-side server proxy. An additional pair is generated for the client-side server proxy if the client-side server proxy needs to regenerate events on behalf of the server bean.
- Select the **Instantiate server object** checkbox if you want to start the RMI registry and the Remote Object Instance Manager. The Remote Object Instance Manager instantiates the server bean through its associated server-side server proxy immediately after the proxies are generated. This is handy if you want to test the remote access connections during development, rather than waiting to test after deployment.
- Select **Finish**.

The generated classes and interfaces now appear in the Workbench package.



Server1
Server1BeanInfo
Server1If
Server1S
Server1S_Skel
Server1S_Stub

Notice that a remote communication icon is beside some of the class and interface names, providing you with a visual reminder that these objects were generated by the RMI Access Builder. The bottom two classes were generated by the RMI compiler (discussed below). The name of each generated class or interface shares a common root, as specified in the **Proxy Bean Name** field of the SmartGuide.

In the example above, the generated code includes:

- *Server1* is the client-side server proxy. This is the main class that implements the proxy bean.
- *Server1BeanInfo* contains information on the client-side server proxy bean interface, enabling you to view and edit the bean features (properties, events, methods) from the BeanInfo page for the class.
- *Server1If* is the server-side server proxy's RMI interface, which defines which server bean public methods are accessible through the client-side server proxy.
- *Server1S* is the server-side server proxy.
- *Server1S_Skel* is the RMI skeleton class that is generated by the RMI compiler, using the server-side server proxy as input.

- *Server1S_Stub* is the RMI stub class that is generated by the RMI compiler, using the server-side server proxy as input.

Creating RMI stubs and skeleton classes

In the Create Proxy Bean SmartGuide, if you had selected the **Create RMI stub and skeleton for generated classes** checkbox, the RMI stub and skeleton code would have already been generated for you. The stub and skeleton code facilitates communication. Whether you choose to generate this code manually or automatically (via the SmartGuide), this is a two-step process:

1. Generating the server proxy and its associated classes and interfaces
2. Generating RMI source and compiled output

You may choose to generate your access classes in two steps if you prefer to use your own RMI source. VisualAge for Java integrates an RMI compiler in the Workbench (compatible with the JDK 1.1 RMI compiler), so even this two-step process is easy:

1. In the Workbench, select the class that represents the server-side server proxy.
2. From the pop-up menu, select **Tools**, then select **Remote Method Invocation** and **Generate Proxies**.
3. If your server bean generates events, you'll need to repeat steps 1 and 2 on the client-side server proxy.

The RMI compiler processes the proxy, generating Java stubs and skeleton code. The new classes are placed in the same package as the proxy classes.

Connecting the client-side server proxy

Now that you've created the proxy bean and its supporting classes and interfaces, you're ready to add the proxy bean (specifically, the client-side server proxy) to your application interface.

Follow these steps in the Workbench:

1. Select the class in the package that represents the user interface, and from the pop-up menu, select **Open To** and then **Visual Composition**.
2. In the Visual Composition Editor, select **Add Bean** from the **Options** pull-down menu.
3. In the Add Bean dialog, click on **Browse** beside the **Class Name** field.
4. In the Choose a valid class secondary window, select the client-side server proxy and select **OK**.
5. On the free-form surface, move the cursor to an area outside the visual section and click mouse button 1. The client-side server proxy is now visually represented.
6. Connect the proxy bean to the interface as required. As seen in previous sections that demonstrate connections, the features of the proxy bean can be accessed.

7. Set proxy bean properties as required.

Now that you've developed all your code, let's discuss some test considerations.

Starting the server

Before you test your code in the Workbench, you need to:

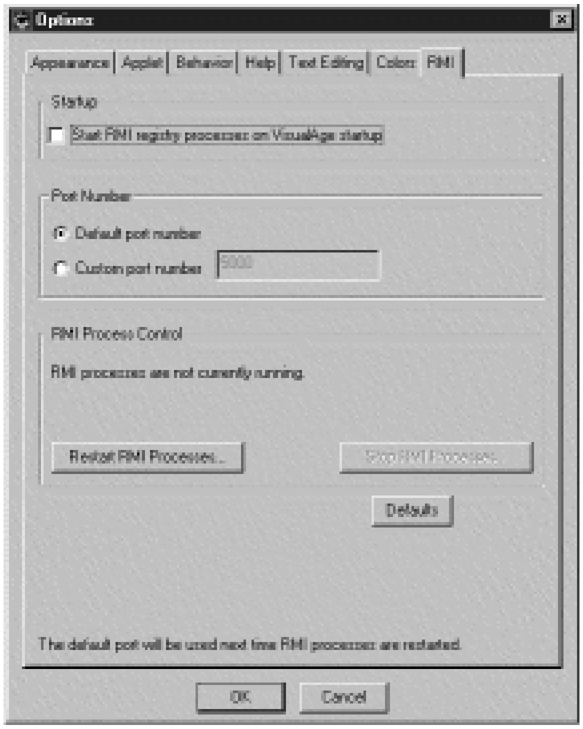
1. Start the RMI registry
2. Start the Remote Object Instance Manager

The RMI registry

The RMI registry is a server-side program that allows clients to get a reference to the server-side server proxy, through which the server bean is accessed. Registry entries are URL-based. Once a server-side server proxy is registered on the server, clients can remotely reference the server-side server proxy by name and remotely invoke its methods.

The Integrated Development Environment lets you specify RMI registry startup and processing status. To see the RMI options that are set in your development environment:

- Select the **Options** menu item from the **Workspace** pull-down and go to the **RMI** page.



Startup

□ Start RMI registry processes on VisualAge startup

Port Number

● Default port number

○ Custom port number 5000

RMI Process Control

RMI processes are not currently running.

The default port will be used next time RMI processes are restarted.

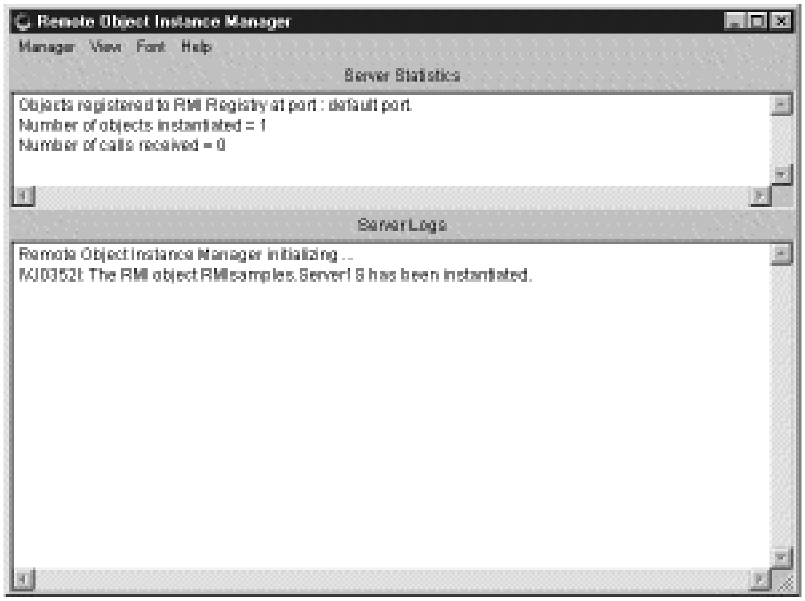
These options let you:

- Indicate if the RMI registry will be started when the Integrated Development Environment is started.
- Specify a port number
- Start and stop RMI processes during development

The Remote Object Instance Manager

The Remote Object Instance Manager is a server program that creates and manages instances of server beans and server-side server proxies.

You can manage the instantiation and removal of server objects through the Remote Object Instance Manager console.



```
Objects registered to RMI Registry at port : default port
Number of objects instantiated = 1
Number of calls received = 0
```

```
Remote Object Instance Manager initializing ...
IVJ0352I: The RMI object RMIsamples.Server1S has been instantiated.
```

The **Server Statistics** pane displays information on the status of calls made to the server objects and the number of instantiated objects. The **Server Logs** pane displays status messages and the names of objects that have been instantiated. Use the menu options to instantiate or remove server objects.

The RMI registry must be started before the Remote Object Instance Manager can be started.

Starting the server programs

Recall that the Create Proxy Bean SmartGuide includes a checkbox labeled **Instantiate server object** which, when selected, automatically starts both the RMI registry and the Remote Object Instance Manager immediately after the creation of the proxy bean and its associated classes and interfaces.

If you didn't select the checkbox, you must start them manually:

- In the Workbench, select the class representing the server-side server proxy.
- From the pop-up menu, select **Tools**, then **Remote Bean Access** and **Instantiate Bean in Server**. The RMI registry and Remote Object Instance Manager are started, and instances of the server bean are created.

Summary

With the VisualAge for Java RMI Access Builder, developing distributed, Java-to-Java, client-server code is simple. The RMI Access Builder generates proxies, beans, and RMI access code in an integrated environment. You just need to supply the server bean that will be distributed, and VisualAge for Java takes over from there.

C++ Access Builder

With the VisualAge for Java C++ Access Builder, you can access your C++ DLLs from your Java applets and applications.

The advantages of accessing your C++ code from a Java client program include:

- You don't need to rewrite your C++ code in Java. Leverage your existing C++ libraries for quicker Java development.
- You can port a C++ application in steps, instead of all at once. This is vital if you have some sensitive or critical C++ code that can't be quickly ported, yet you still need to provide flexible Java access fast.
- You can code features that are not currently available in the Java language.
- Speed. A constant (and sometimes unfair!) complaint about Java is that what you gain in portability you lose in performance; platform-independent bytecodes can't run as fast as optimized native binary code. Some C++ applications, particularly server applications, need to run fast. Write performance-sensitive parts of your program in C++, and still take advantage of Java's flexibility and ease-of-use.

In this section, we'll review the high-level steps required to access C++ code from Java:

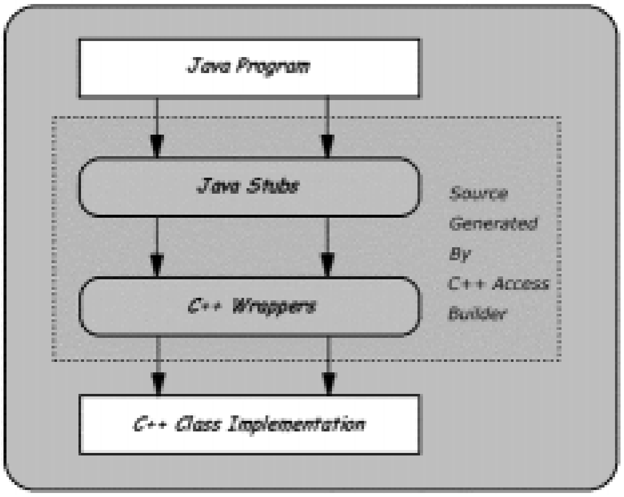
- Generating the C++ wrapper and stub beans
- Working with the generated makefile
- Importing the stub beans into the IDE
- Distributing and connecting the stub beans

Generating the C++ wrapper and stub beans

The C++ Access Builder creates a bridge between Java and C++ by creating:

- A C++ wrapper for each accessible class in your C++ DLL
- A JavaBeans stub class that corresponds to each C++ wrapper

- A makefile that you can use to compile both the generated C++ source and Java source



To construct the C++ wrapper, you need to provide two essential pieces of information to the C++ Access Builder:

- A base name, used to specify the name of the shared library and other files used to build the library
- Header files, which contain definitions of all the C++ classes you wish to access from Java. For example:

```
class Rect: public Shape
         {
  public:
  Rect(double width, double length);
  virtual double area();
  private:
  double _width;
  double _length;
};
```

This header file is used as input in the example below.

The name of the generated DLL is not required. In fact, the DLL doesn't even need to exist; the generated makefile (discussed later) includes commands to create the DLL based on the header file information.

You can also optionally provide the following information:

- A Java package name for the stub beans (recommended if the stub beans will be imported into the Integrated Development Environment)

- C++ preprocessor and compiler commands to process the supplied header files
- A target directory
- C++ source files used to generate the DLL you want to wrap
- Shared libraries to link with the DLL

Unlike the other Enterprise Access Builders, the C++ Access Builder is run from the command line. The ivj2cpp command invokes the C++ Access Builder from the command line. Enter ivj2cpp `-h` on the command line to see the tool options. Here's an example of invoking the C++ Access Builder:

```
ivj2cpp myLib Rect.hpp -p myServerApp -c "icc /Pd"

  -s Rect.cpp -d c:\output
```

where

- ivj2cpp invokes the C++ Access Builder
- *myLib* is the base name
- *Rect.hpp* is the header file
- *myServerApp* is the name of the package
- "icc /Pd" invokes the VisualAge for C++ compiler and preprocessor to preprocess myClass.hpp. The /Pd preprocessor option redirects output to stdout.
- *Rect.cpp* is the C++ source file used to generate the DLL. The source file is specified in the generated makefile.
- "c:\output" is the directory where the generated C++ wrappers and stub beans will be placed

In this example, the C++ Access Builder generates the following files:

- *Rect.java*
- *RectWrapper.cpp*
- *Makefile*
- *myLib.mk*

The generated bean includes declarations for the native methods, as well as a static load of the required C++ library.

```
/*

 * This file was generated by the IVJ2CPP tool.
 * DO NOT EDIT THIS FILE.
 */
package myServerApp;

import COM.ibm.ivj.eab.j2cpp.*;

public class Rect extends Shape {
  ...
  private native double area_p();
  public double area() {
  return area_p();
  }
  ...
  static {
    System.loadLibrary("myLib");
  }
/*
 * End of file generated by the IVJ2CPP tool.
 */
```

Notice that only the declarations for the C++ methods are included here. The definitions, of course, appear in the C++ DLL. The loadLibrary method call specifies the DLL base name as a parameter.

The generated C++ wrapper code uses the Java Native Interface (JNI) to communicate with the stub beans.

Working with the generated makefile

The C++ Access Builder also generates a makefile that includes commands to:

- Compile the generated C++ wrappers with the specified C++ preprocessor and compiler commands. The default compiler used is the IBM VisualAge for C++ compiler, although you can also use the Microsoft Visual C++ tools.
- Compile the generated stub beans using the javac compiler provided with the JDK. If you subsequently import the beans into the Integrated Development Environment (as we'll demonstrate), this compilation is not necessary; we've already seen that the IDE compiles your source for you.
- Create a new DLL that combines the original library with the generated code.

Of course, you are not obligated to use the makefile as is. You can modify it or incorporate parts of it into existing makefiles. You will need a makefile processing tool, like the VisualAge for C++ NMAKE tool, to process the makefile, or you can manually enter the compile and link commands.

Using the C++ Access Builder is this simple. Now you're ready to integrate the stub bean with your Java client. But first, let's go through a brief discussion of deployment scenarios.

Local and remote access scenarios

Do you plan to access the C++ DLL from a Java applet or application? Do you want to minimize the requirements on the client machine or download time? You can't download a DLL at the same time that you're viewing an applet on a web page. And because of applet security restrictions, a web page applet cannot access a DLL that doesn't reside on your local machine.

Two common scenarios for accessing the DLL include the local and remote scenarios:

Local scenario

Imagine a scenario where end-users access an application through a common, platform-independent front-end (written in Java), and some of the business logic is written in C++ for performance benefits or because you want to leverage your existing C++ code base. The Java client application and C++ DLL both reside on the local machine.

However, perhaps you don't want to trouble end-users with installing this code locally. Each operating system would require a different flavor of the DLL. You'd prefer to let the user access your code across the web, requiring only a web browser.

Remote scenario

To avoid the applet security restrictions, you can distribute the stub bean in a two-tiered model. The stub bean and C++ DLL reside on the server, connected to the client through a proxy bean that accesses the remote stub bean using RMI services. The stub bean becomes the distributed server bean.

Note that if the stub bean references non-primitive objects, you need to create a serialized interface for the stub bean. In this case, you will distribute the serialized interface in the two-tiered model, not the stub bean.

With the C++ code and run-time code on the server, you don't need to worry about accommodating multiple platforms; the downloadable client code is all Java.

Using the RMI Access Builder discussed in “RMI Access Builder” on page 132, the VisualAge for Java suite of tools lets you quickly put together this remote access scenario.

Importing the beans into the IDE

After creating the Java-C++ bridge (stub bean and C++ wrapper), you're ready to import the bean into the Integrated Development Environment for use with the rest of your Java program. The following steps show you how you would bring it all together:

- In the Workbench, create a new project or select the project that contains the Java client.

- From the **File** pull-down menu, select **Import**. Within the Import Type SmartGuide, specify the name of the class source (`.java`) you'd like to import. The C++ Access Builder creates a Java package specification to contain the stub class. When the stub source file is imported, the package is automatically created. If you had to create a serialized interface for the stub bean, you must import that class as well.

Of course, when you imported the classes into the IDE as working editions, open editions of the classes were compiled and saved in the repository.

Distributing and connecting the stub bean

Now that the stub bean (or serialized interface) is in the Integrated Development Environment, we can distribute it for remote access. Follow the same set of tasks as outlined in the previous section on RMI:

- Select the stub bean (or serialized interface), and from the pop-up menu, select **Tools**.
- Select **Create Proxy Beans** to open the SmartGuide for the RMI Access Builder and fill in the fields, ensuring that the stub bean (or serialized interface) is specified in the Class Name field. RMI source classes are generated.
- Use the RMI compiler (also from the **Tools** menu) to generate the RMI stubs and skeletons.

To wire a bean into the client interface, go to the Workbench:

1. Select the class in the package that represents the user interface, and from the pop-up menu, select **Open To** and then **Visual Composition**.
2. In the Visual Composition Editor, select **Add Bean** from the **Options** pull-down menu.
3. In the Add Bean dialog, click on **Browse** beside the **Class Name** field.
4. In the Choose a valid class secondary window, select the client-side server proxy (created in the steps above on RMI) and select **OK**.
5. On the free-form surface, move the cursor to a nonvisual bean area and click mouse button 1. The client-side server proxy of the stub bean is now visually represented.
6. Connect the proxy bean to the interface as required. As seen in previous sections that demonstrate connections, the features of the stub bean (or serialized interface) can be accessed.

That's it! By following these steps, you'll be able to create Java programs that leverage your existing C++ code.

Chapter 9. More information about VisualAge for Java

This Getting Started document is only a brief overview of what you can do with VisualAge for Java. For more complete information, see the complete set of online help that is available from the **Help** menu of any window in VisualAge for Java.

This online help is organized into three categories, all of which are directly accessible both from the home page of the Help and from any of the content pages:

- **Concepts** - definitions and overall grounding in the concepts you need to know to use VisualAge for Java
- **References** - operational details and other kinds of reference information organized to make it easy for you to retrieve what you need
- **Tasks** - how to perform tasks: step-by-step guidelines for accomplishing specific goals

Printing material

You can print any topic in the help for VisualAge for Java. To print a topic:

1. Display the help topic you want to print.
2. Select the content frame (the bottom-right frame) by clicking mouse button 1 on the frame.
3. Select **Print Frame** from the **File** menu.

Index

A

adding beans to the free-form surface 14
aligning beans 15, 60
applet
- creating 11

applications
- running 50

arranging connections 76

B

BeanInfo classes 105
BeanInfo page 105, 107
- creating a new property 107

beans 5, 104
- adding to free-form surface 44
- aligning 15, 16, 42, 60
- BeanInfo classes 105
- Buttons 15
- changing properties 62
- changing text 14
- color portability 65
- colors 63
- connecting 17, 65
- connecting buttons 47
- copying 59
- deleting 59
- deselecting 58
- displaying pop-up menus 12
- distributing evenly 61
- dragging 12
- features 104
- file dialogs 45
- font portability 65
- fonts 64
- Labels 14
- Lists 14
- manipulating 57
- matching dimensions 61
- matching width 16
- moving 58
- multiple selection 58
- Properties window 62
- saving 17, 20
- selecting 57

beans *(continued)*
- selecting more than one 12
- setting properties 26
- sizing 15, 60
- testing 20
- Text 14

beans palette 5, 14
Breakpoints 93, 101
Breakpoints window 102
browsing 97
- classes 99
- interfaces 99
- methods 100
- packages 98
- projects 97

browsing connections 75

C

C++ Access Builder 139
- connecting the stub bean 144
- distributing the stub bean 144
- generating stub beans 139
- generating the C++ wrapper 139
- importing beans into the IDE 143
- local access scenario 143
- remote access scenario 143
- working with the generated makefile 142

changing bean properties 62
changing the color of a bean 63
changing the default printer 90
changing the font of a bean 64
changing the text of a bean 14
CICS Access Builder 127
- connecting the CICS access beans 131
- Create COMMAREA Bean SmartGuide 129
- overview 128
- Unit of Work bean 131

classes 1
- browsing 99
- creating 33

comparing editions 86
connections 5, 17, 27
- arranging 76
- changing endpoints 76
- changing properties 72

connections *(continued)*
- deleting 74
- event-to-method 17, 67
- event-to-script 69
- hiding 75
- parameter connections 69
- parameter-from-property 17
- parameters 72
- property-to-property 66
- reordering 75
- showing 75
- To-Do File program 47

Console 91
copying beans 59
correcting mistakes 17
Create Class SmartGuide 33
creating a new class 33
creating a new edition 82
creating methods 27, 35
customizing the workspace 112

D

Data Access Builder 117
- connecting to your own GUI 126
- customized SQL statements 122
- customizing the schema mapping 121
- distributing and deploying the GUI 127
- generating data access beans 123
- generating the schema mapping 119
- stored procedure calls 122
- thin-client model 118
- using the generated GUI 125

Debugger 92
Debugger window 102
debugging 100
- removing breakpoints 101
- setting breakpoints 101
- using the Debugger window 102

deleting beans 59
deleting connections 74
deselecting beans 58
distributing beans evenly 61

E

editions 2, 24
- managing 79

Enterprise Access Builders xiii, 117
- C++ Access Builder 139

Enterprise Access Builders *(continued)*
- CICS Access Builder 127
- Data Access Builder 117
- RMI Access Builder 132

events 5
examples in the repository 87
exporting code 3

F

features 104
file dialog beans 45
file input and output 31
finding program elements 23
free-form surface 5
- adding beans 44

H

hiding connections 75

I

importing code 3
installation 9
interfaces 1
- browsing 99

J

JavaBeans 5, 104
JDBC 117

L

Log 91

M

managing editions 79
- comparing editions 86
- creating a new edition 82
- examining examples 87
- exploring the repository 85
- returning to a previous edition 85
- versioning 81

manipulating beans 57
matching bean dimensions 61
methods 1, 5

methods *(continued)*
  browsing 100
  creating 27, 35
moving beans 58
moving between windows 90

N

navigating 90

O

online help 145
open editions 80
Options window 112

P

package
  creating 11
packages 1
  browsing 98
primary selection 58
printing
  changing the default printer 90
  material from the online help 145
printing program elements 89
program elements 1, 2
  browsing 97
  printing 89
  searching 94
projects 1
  browsing 97
  creating 11
properties 5, 26, 62
  enable 27
Properties window for beans 62
proxy bean 134

Q

Quick Start 10

R

rapid application development 1
readToDoFile method 49
redo 17
removing breakpoints 101
reordering connections 75
repository 2, 79
Repository Explorer 85, 92
returning to a previous edition 85
RMI Access Builder 132
  connecting the client-side server proxy 136
  Create Proxy Bean SmartGuide 134
  creating RMI stubs 136
  creating skeleton classes 136
  overview 133
  Remote Object Instance Manager 138
  RMI registry 137
  starting the server programs 138
running applications 50

S

samples
  adding state checking 23, 25
  BeanInfo version of To-Do File program 107
  in the repository 87
  To-Do File program 31
    behavior 31
    creating a new class 33
    steps to create 32
  To-Do List applet 9, 13
  To-Do List counter 81
saving 20, 30
saving the workspace 21
saving your work 17
schema mapping
  customizing 121
  generating 119
Scrapbook 41, 91
searching 94
  Search window 94
  Workspace menu 96
selecting beans 57
selecting beans from the beans palette 14
selection handles 57
setting breakpoints 101
setting options 112
setting properties 26
showing connections 75
sizing beans 15, 60
SmartGuides 1
  Create Applet 11
  Create Class 33
  Create COMMAREA Bean 129
  Create Method or Constructor 27

SmartGuides *(continued)*
- Create Proxy Bean 134
- Data Access Builder 119
- Method Properties 35
- Versioning Selected Items 24, 55

source of a connection 5
starting VisualAge for Java 9
state checking 23
- saving and testing 30

status area 5

T

target of a connection 5
testing 20, 30, 41, 50
To-Do File program
- adding buttons 42
- adding file dialog beans 45
- adding the ToDoFile class 44
- BeanInfo version 107
- behavior 31, 32
- connecting beans 47
- connecting buttons 51
- creating a new class 33
- creating a skeleton class 33
- disposing of the file dialog 48, 52
- invoking readToDoFile 49
- invoking writeToDoFile 53
- readToDoFile method 35
- saving 55
- showing the file dialog 48, 52
- testing 50, 55
- ToDoFile class 33
- using the Scrapbook 41
- writeToDoFile method 38

To-Do List applet 9
To-Do List counter 81
tool bar
- Visual Composition Editor 5
- Workbench 3

U

undo 17

V

version control 2
versioned editions 80
versioning editions 24, 81
Visual Composition Editor 5
- adding beans to the free-form surface 44
- aligning beans 15, 60
- bean colors 63
- bean fonts 64
- changing bean properties 62
- connecting beans 65
- connection parameters 72
- connection properties 72
- copying beans 59
- deleting beans 59
- deselecting beans 58
- detailed description 57
- distributing beans evenly 61
- event-to-method connections 67
- event-to-script connections 69
- manipulating beans 57
- manipulating connections 74
- matching bean dimensions 61
- moving beans 58
- multiple selection 58
- opening 25
- parameter connections 69
- portability of colors and fonts 65
- Properties window 62
- property-to-property connections 66
- saving your work 17
- selecting beans 57
- sizing beans 15, 60

VisualAge for C++ 142

W

where to find more information 145
windows
- Breakpoints 93, 102
- Console 91
- Debugger 92, 102
- Log 91
- Options 112
- Quick Start 10
- Repository Explorer 85, 92
- Scrapbook 41, 91
- Search 94
- Workbench 10

Workbench 3, 10
- finding program elements 23

workspace 2, 79
- saving 21

Part Number: 4304086

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.