IBM VisualAge for Java, Version 2.0

Getting Started

S430-4086-01

IBM

IBM VisualAge for Java, Version 2.0

Getting Started

S430-4086-01

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

Second Edition (August 1998)

This edition applies to Release 2.0 of IBM VisualAge for Java 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 your 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
3G / 950 / 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

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries 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 IBM's intellectual property rights 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 user's responsibility.

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:

Director of Licensing
Intellectual Property and & Licensing
International Business Machines Corporation
North Castle Drive, MD - NC119
Armonk, New York 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

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

Trademarks and Service Marks

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

Chapter 1. Introduction

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 these sections:

- The basics introduces the overall capabilities of VisualAge for Java and outlines the concepts you need to know.
- Building your first applet introduces the visual programming features of VisualAge for Java by leading you through the process of creating a simple applet.
- Adding state checking to your applet 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.
- Creating the To-Do List program shows you how to add more features to your applet and gives you more details on VisualAge for Java's overall coding environment.
- What else you can do with the Visual Composition Editor gives you more details on the powerful visual programming capabilities of VisualAge for Java.
- Managing editions of your program gives you more details on the edition control features of VisualAge for Java.
- What else can you do with the IDE gives you more information on the following features of VisualAge for Java:
  - Printing
  - Navigating
  - Searching
  - Browsing
  - Debugging

  - Support for JavaBeans
  - Customizing your programming environment
- Domino AgentRunner introduces you to building, running and debugging Domino agents in VisualAge for Java.
- Where to find more information describes the overall help system that comes with VisualAge for Java. It also gives you details on printing material from the help system.

Sample program in this document

By doing the exercises in this document, starting with Building your first applet, you will create a sample program called 'To-Do List'. 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.

See Examining examples in the repository for details on how to examine the completed version of this example.

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 a sample program. 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. 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.

What’s new in VisualAge for Java Version 2.0

VisualAge for Java, Version 2.0 includes the following new features:

- Support for JDK1.1.6, including Swing 1.0.2, inner classes and anonymous classes, and the Java Native Interface (JNI)
- New IDE features

- Advanced coding tools such as automatic formatting, automatic code completion and fix-on-save
- Context-sensitive help
- Advanced debugging tools such as conditional breakpoints and both multiple and incremental program debug
- Support for JavaDoc output

- New Visual Composition Editor features
  - Visual programming support for Swing beans
  - Wizards for string externalization to assist in building multi-language applications
  - Support for object serialization
  - Ability to import GUIs built in other Java IDEs
- New Data Access Beans that provide access to relational data
- For Windows users, the ability to check VisualAge for Java code in or out of VisualAge TeamConnection™, ClearCase, or PVCS
- Tool Integrator APIs that allow you to add third-party tools that are launched from within the IDE, store and retrieve components from the integrated repository, or add JavaBeans to the Visual Composition Editor's parts palette

Where you can get the latest VisualAge for Java information

To get the latest information updates, bookmark this Web site.

www.software.ibm.com/ad/vajava

The *Library* section provides additional Java programming books, papers and links.

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 2. 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** you point and click to:

- Design the user interface for your program
- Specify the behavior of the user interface elements
- Define the relationship between the user 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. 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, called a repository. VisualAge for Java shows code 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 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 currently 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.

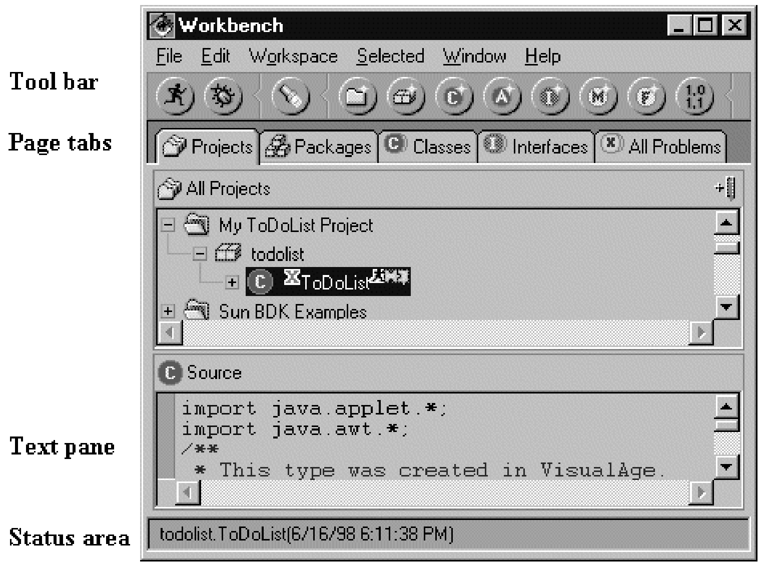
Importing and exporting code

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.



```
import java.applet.*;
import java.awt.*;
/**
 * This type was created in VisualAge.
```

todolist.ToDoList(6/16/98 6:11:38 PM)

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. Specifically, on the Projects page, from left to right on the tool bar, the tools are: run, debug, search, create program elements such as projects and packages, and show edition information such as 1.0 or 1.1.

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 **All 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 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 **nonvisualbean** 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.

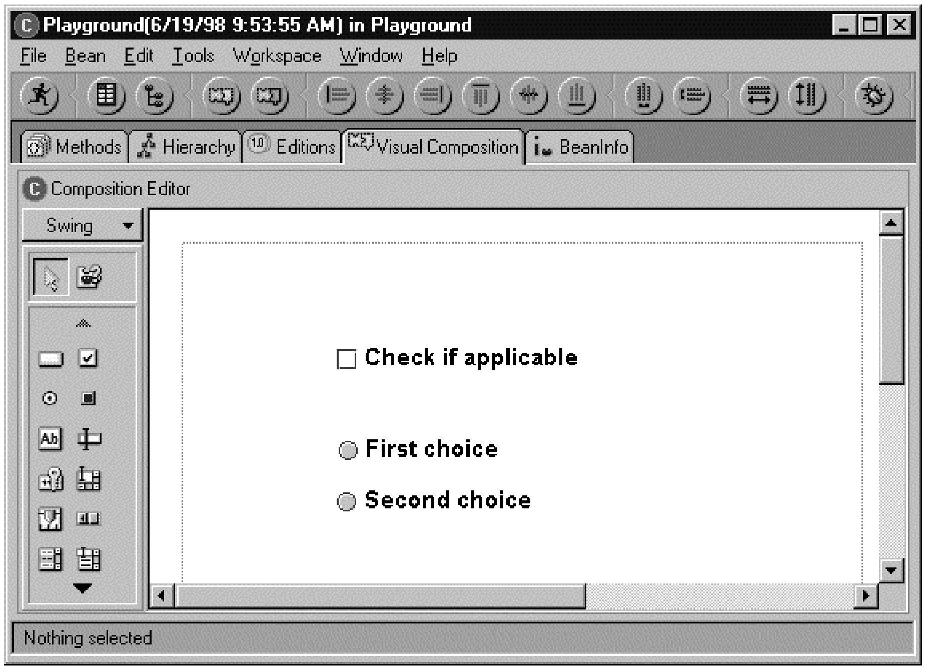
Connections

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.

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. In the diagram below, three beans are on the surface: a checkbox bean and two radio button beans.



Beans palette Free-form surface

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 **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 by placing the mouse pointer over the icon for the 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. Specifically, the tools from left to right are: run, specify properties, provide a beans list, show or hide the connections, arrange the beans on on the surface in a number of ways, and debug.

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

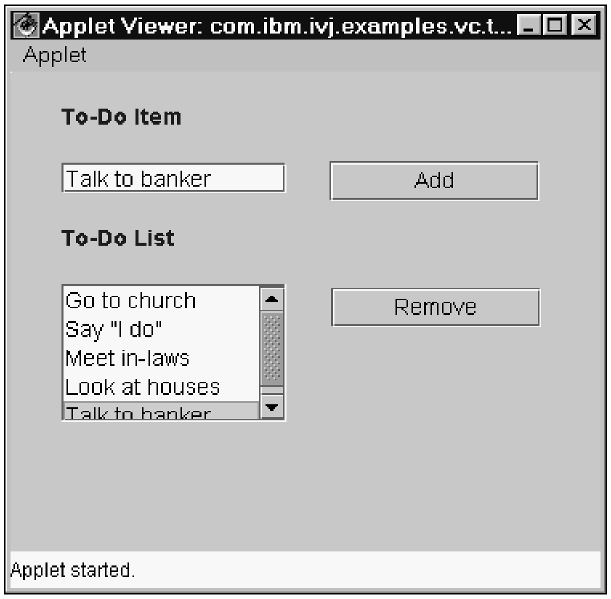
The large open area in the Visual Composition Editor is called the **free-form surface**. You use the free-form surface as the visual programming area where you construct your program. You cannot drop nonvisual beans on top of visual beans.

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 3. Building your first applet

This section guides 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 JTextField bean for entering a To-Do item and a JList bean for displaying the To-Do items. There are also two JButton 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 `readme.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

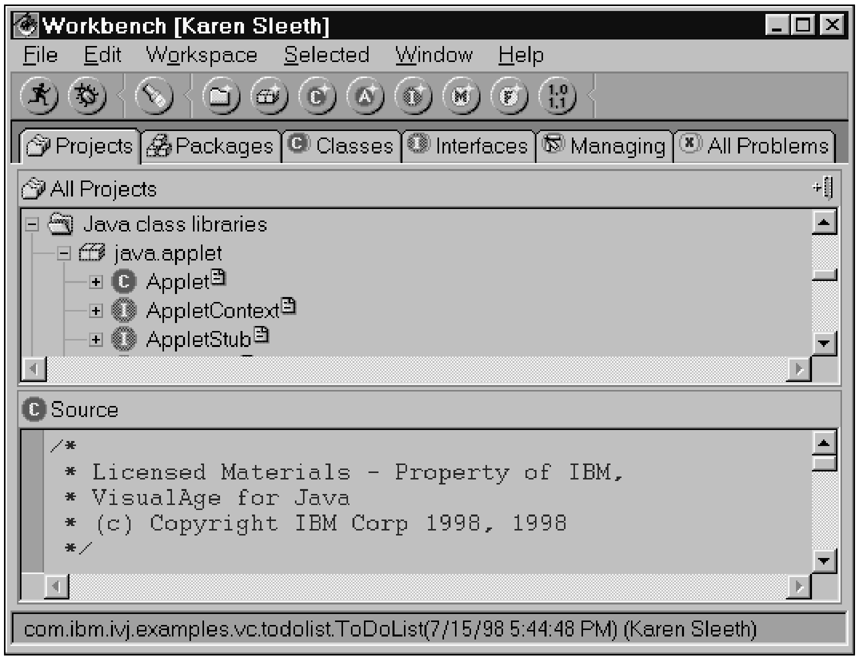
You can start VisualAge for Java by doing one of the following:

- For OS/2®, from the VisualAge for Java folder, double-click the **IDE** icon.
- For Windows 95 and Windows NT, select **VisualAge for Java** and then **IDE** from the **Start - Programs** menu.

[ENTERPRISE] When you first start VisualAge for Java, Enterprise Edition, you will be prompted to choose an owner for your workspace and a network name for the user called Administrator. For the purposes of these exercises, you can select **Administrator** as your workspace owner and you can enter any value as the Administrator's network name. For more information on the team programming environment, see *Getting Started* for VisualAge for Java, Enterprise Edition.

Now that you are up and running

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

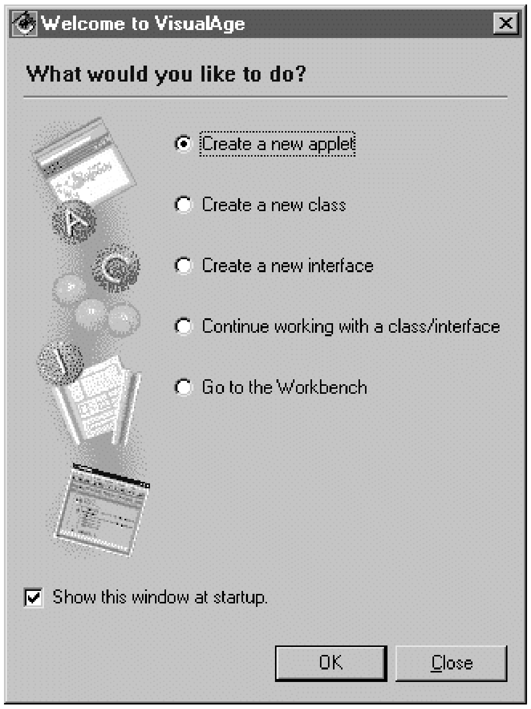


```
/*
 * Licensed Materials - Property of IBM,
 * VisualAge for Java
 * (c) Copyright IBM Corp 1998, 1998
 */
```

com.ibm.ivj.examples.vc.todolist.ToDoList(7/15/98 5:44:48 PM) (Karen Sleeth)

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

Next, the VisualAge Welcome window appears:



The VisualAge Welcome window provides a fast path to creating applets, classes, and interfaces.

Choose **Go to the Workbench** and click **OK**.

Tip: Another useful window that helps you accomplish common tasks quickly and easily in the IDE is the Quick Start window. It includes tasks for creating program elements, learning to use the Scrapbook, managing the repository, and adding samples and useful beans to your workspace. You can launch it any time by pressing F2 or selecting **QuickStart** from the **File** menu in any IDE browser. Keep this in mind when you are creating and managing programs of your own in the IDE. See Using the Quick Start window for more information.

Using a SmartGuide

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

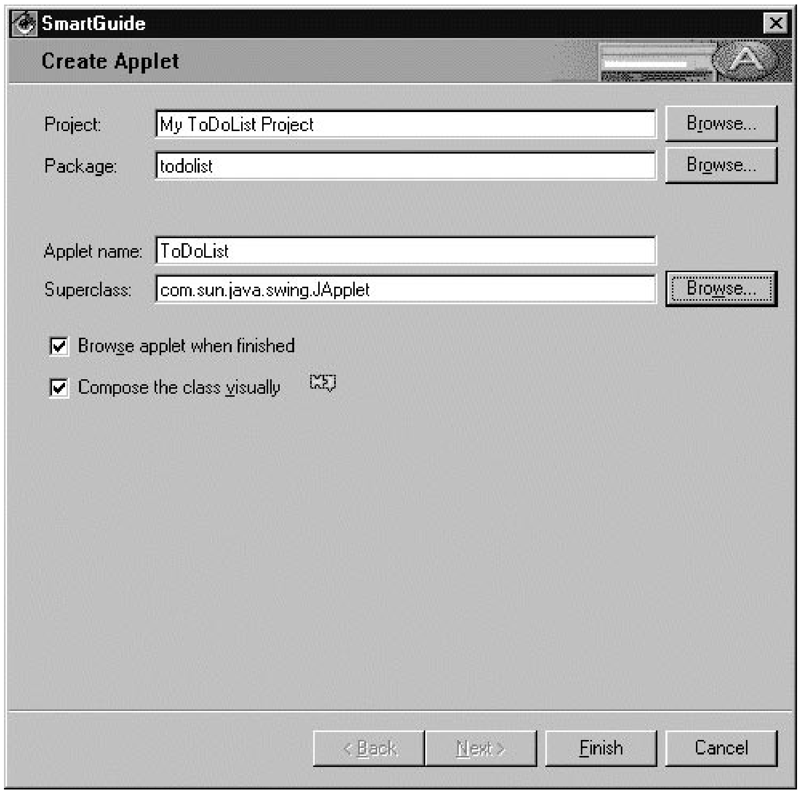
When you manipulate your new applet in the Visual Composition Editor, you are visually manipulating JavaBeans. These JavaBeans (or, simply, *beans*) are represented as classes when you examine your applet in the Workbench.

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 Workbench **File** menu, select **Quick Start**. Then from the Quick Start window:

1. Select **Basic** and then **Create Applet**.
2. Select **OK**. The Create Applet SmartGuide opens.



Project: My ToDoList Project
Package: todolist
Applet name: ToDoList
Superclass: com.sun.java.swing.JApplet
[✓] Browse applet when finished
[✓] Compose the class visually

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

1. In the **Project** field, type a project name, such as *My ToDoList Project*.

2. In the **Package** field, type a package name, such as *todolist*.
3. In the **Applet name** field, type a name, such as *ToDoList*.
4. In the **Superclass** field, using Browse, select **JApplet**. It becomes *com.sun.java.swing.JApplet*.

   Do not use Applet (*java.applet.Applet*) as it is used with the Abstract Windowing Toolkit (AWT). The applet you are building uses Swing components. As a rule, Swing beans should be used with the JApplet superclass; AWT beans should be used with the Applet superclass.
5. Select **Compose the class visually**.
6. Select **Finish**.

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 containing several visual beans.

Working with beans

When working with beans in VisualAge for Java, you 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 crosshair to the desired location and release the button.

Selecting multiple beans

To select multiple beans, hold down the 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 cannot select the window bean and one of the beans it contains together.

Displaying bean pop-up menus

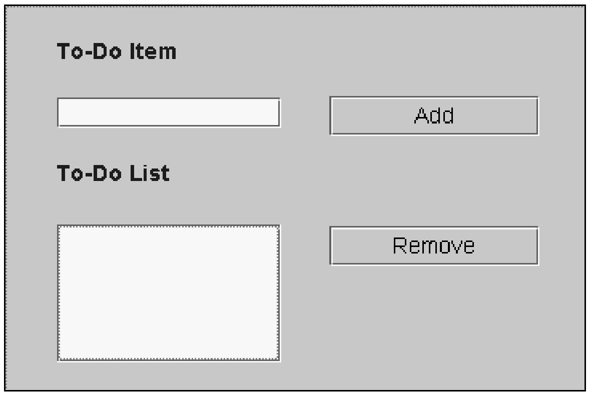
To display a pop-up menu, click mouse button 2 on the bean.

Are You Familiar with Event-to-Code Programming?

The Visual Composition Editor is a visual programming environment. By using icons and mouse actions, you create your application code. However, you can also link visual events, such as clicking a button, with code that you yourself create. See the Event-to-Code Connection section in Connecting Beans, to see how to access hand-coded methods through an event-to-code feature.

Building the To-Do List applet

When the Visual Composition Editor opens for the new ToDoList applet, the default JApplet is represented as a gray 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, size, and align the remaining beans.

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

Although this bean uses a <null> layout manager, it could be built using one of the layout managers, such as GridBagLayout. For an example of creating the user interface for this bean in GridBagLayout, see 'Creating a GUI Using GridBagLayout' in the online helps.

Adding a text field and a label

In this stage of the applet creation, you add a JTextField bean that is used to enter the To-Do items, and a JLabel bean to identify the field. These beans are in the Swing category of the beans palette.

Tip: As you work with the palette, you may find the icons too small for your preference. To make them larger, right-click on any gray area on the palette and select **Show Large Icons** from the pop-up pane.

1. From the beans palette, select the **JTextField** bean. The information area at the bottom of the Visual Composition Editor displays *Category: Swing Bean: JTextField*, reflecting the current selection in the beans palette.
2. Move the mouse pointer over the JApplet bean (the rectangle on the free-form surface). The pointer changes to a crosshair, indicating that it is now loaded with the bean you selected. Click mouse button 1 where you want to add the JTextField.

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

   After you have added the JTextField bean to the JApplet, you can move it to a new location by dragging it with the mouse. You can also resize it by dragging a side of the rectangle.
3. Select the **JLabel** bean and add JLabel just above the JTextField.

   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 and adding another label

Change the text of JLabel to `To-Do Item` by editing the text as follows:

1. Select the JLabel bean. Select **Properties** from the tool bar.
2. In the Properties window, click text field, and type `To-Do Item` (instead of JLabel1). The label now contains the text `To-Do Item`. You may need to stretch the label to see it.
3. You can add another label below the JTextField by copying JLabel1. To copy a bean:

   a. Select the bean.
   b. From the **Edit** menu, select **Copy**.
   c. From the **Edit** menu, select **Paste**. The pointer changes to a crosshair, indicating that it is now loaded with the bean you selected.
   d. Click mouse button 1 below the JTextField to add the new label.
4. Change the text for the new label to `To-Do List` by editing it as you did for JLabel1.

Tip: You can also copy a bean using the Ctrl key. Position the mouse pointer over the JLabel bean, hold down the Ctrl key, and drag the copy of the bean to below the JTextField bean.

Adding a scroll pane for your list

Add a scroll pane so that your list of items can be scrolled.

1. Select the **JScrollpane** bean. (Notice it comes from another group on the palette; the group of container beans.)
2. Add JScrollPane below the text field.

Adding a list

To create the list in which the To-Do items are displayed, you need to add a JList:

1. Select the JList bean and place it inside the scroll pane. The JList bean adjusts to fill the JScrollPane.
2. In the Properties window, change the *selectionMode* property to SINGLE_SELECTION. This simplifies the code needed to handle selection within the list.
3. It is good practice to save your work periodically. To save your work, from the **Bean** menu select **Save 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 instance of a bean at a time, press and hold the Ctrl key before selecting the bean.
2. Select the **JButton** bean. Add a button to the right of the text field.

   Notice that the mouse pointer remains a crosshair, indicating that it is still loaded with the JButton bean. To add another JButton, click mouse button 1 below JButton1.

3. Select the **Selection** tool from the beans palette to unload the mouse pointer.
4. Change the text on JButton1 to `Add` and JButton2 to `Remove` by editing them as you did with the label beans.
5. Save your work using **Save Bean** from the **Bean** menu.

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

Sizing and aligning visual beans

Since this bean does not use a layout manager, you need to clean up the appearance of your user interface by using 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.

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 text field, do the following:

1. Select the Beans List from the tool bar.

   From the list, select JScrollPane1. Remember, you want to size the container the list is in, which is the scroll pane
2. Hold down the Ctrl key to select multiple items and select the text field using mouse button 1.
3. Select the **Match Width** tool from the tool bar.

   Because the text field was selected last, it becomes the anchor bean for the match width operation. The width of the list is changed to match the width of the text field.

Note: The anchor bean 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 text field, list, and labels, do the following:

1. Move the label for the text field (the one that says **To-Do Item**) to the position you want it in the applet.
2. Select the scroll pane (making certain it is the scroll pane, not the list), the text field and and their associated labels, making sure to select the label of the text field last.

   By selecting the text field label last, you make it the anchor bean for the alignment operation.
3. Select the **Align Left** tool from the tool bar.
4. Because the text field, list, and labels are still selected, you can evenly distribute them in the window by selecting the **Distribute Vertically** tool from the tool bar.
5. Save your work.

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

Note: The entire applet that you are creating is a bean. When you select **Bean** and then **Save Bean** from the 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 changes.

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 add the text entered in the text field to the list when the **Add** button is selected, and to remove a selected item from the list when the **Remove** button is selected. To do this, you need to make **event-to-method**connections between the buttons and the text field and the list. In the example, you will extend your list to include a model called DefaultListModel. The Java Foundation Classes, also known as Swing, separate data from the view of the data. The actual list items are stored in the list model and then a connection sends the model's data to the list in the applet.

Because selecting a button signals an *actionPerformed(java.awt.event.ActionEvent)* event and adding an item to the list model is performed by the *addElement(java.lang.Object)* method, the event-to-method connection for adding an item to the list model is between the **Add** JButton's *actionPerformed(java.awt.event.ActionEvent)* event and the DefaultListModel's *addElement(java.lang.Object)* method. Removing an item from the list is performed by connecting the **Remove** JButton's *actionPerformed(java.awt.event.ActionEvent)* event to the *removeElementAt(java.lang.Object)* method of the DefaultListModel.

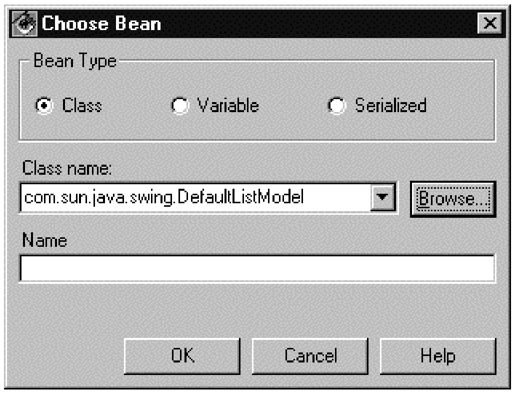
Simply adding these two event-to-method connections does not actually cause anything to be added to or removed from the list model because both the *addElement(java.lang.Object)* and *removeElementAt(java.lang.Object)* methods require a parameter that specifies what object is to be added to or removed from the list. You specify the parameters by creating **parameter-from-property** connections.

The text from the JTextField is provided as the parameter to the *addElement(java.lang.Object)* method. The s*electedValue* from the JList is provided as the parameter to the *removeElementAt(java.lang.Object)* method.

Adding a list model and making your connections

You must first add a list model before connecting your beans. Why do you need a list model? In the Java Foundation Classes (also called Swing) data and views of the data are separate. A list is simply one view of some To-Do items. The To-Do list items themselves are contained in the list model. To add a list model, do the following:

1. Select the **Choose Bean** tool from the palette.
2. Select **Class** as the Bean Type from the Choose Bean window. Using **Browse** with a pattern of 'de', select the **DefaultListModel** class:

   
   Choose Bean

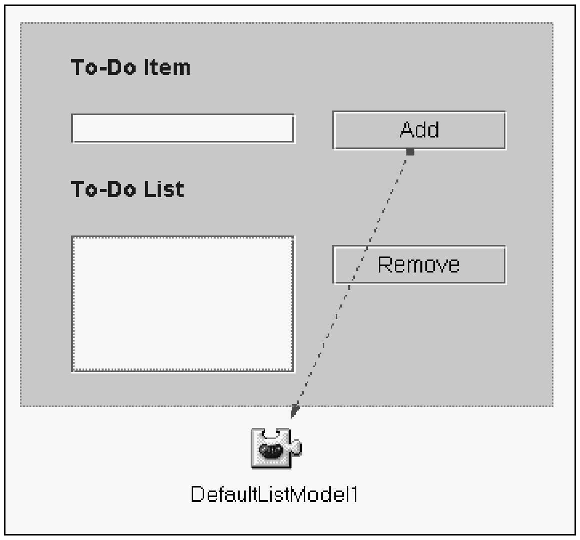
   Bean Type: [✓ Class] Variable Serialized

   Class name: com.sun.java.swing.DefaultListModel

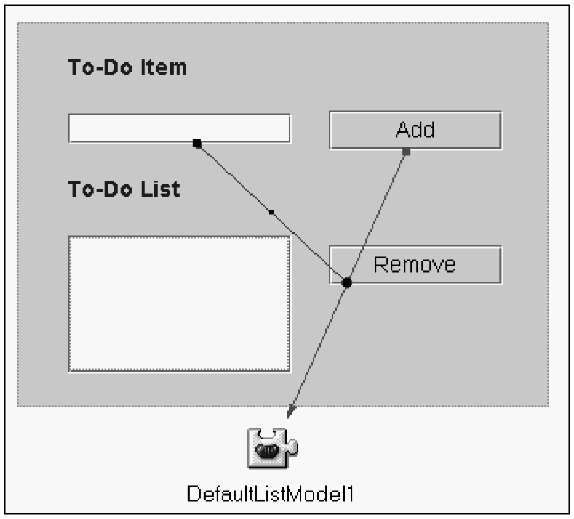
   Name:

   Click **OK** and place the DefaultListModel on the free-form surface, below the applet (the gray area).
3. Now you can begin your connections to move items into your list. Select the **Add** button, then click mouse button 2. In the pop-up menu that appears, select **Connect** and then **actionPerformed**.

   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.
4. To complete the connection, click mouse button 1 on the DefaultListModel and then select **Connectable Features**. Click **addElement(java.lang.Object)** 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 *addElement(java.lang.Object)* method is missing.

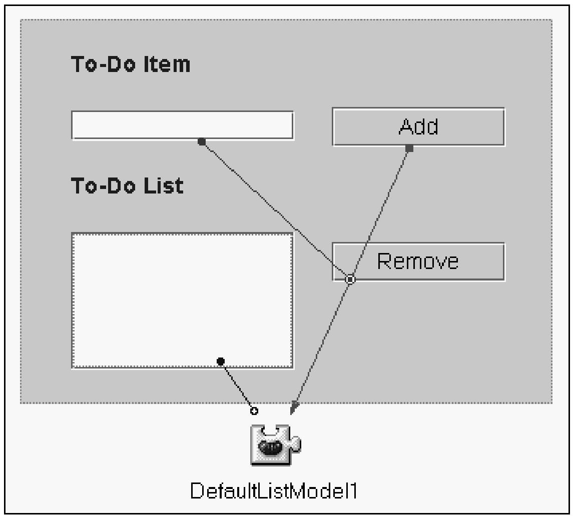


5. To make the parameter-from-property connection that specifies what to add to the list, follow these steps:
   a. Move the mouse pointer on top of the dashed event-to-method connection line.
   b. Click mouse button 2. Select **Connect** then **obj** from the pop-up menu that appears.
   c. Click mouse button 1 on the text field, and then select **text** from the pop-up menu that appears.



6. Finally, you must get the To-Do items from the DefaultListModel to the list in your applet, so that users can see them in the applet. This is a connection that sends data from a source (the model) to a view (the list).

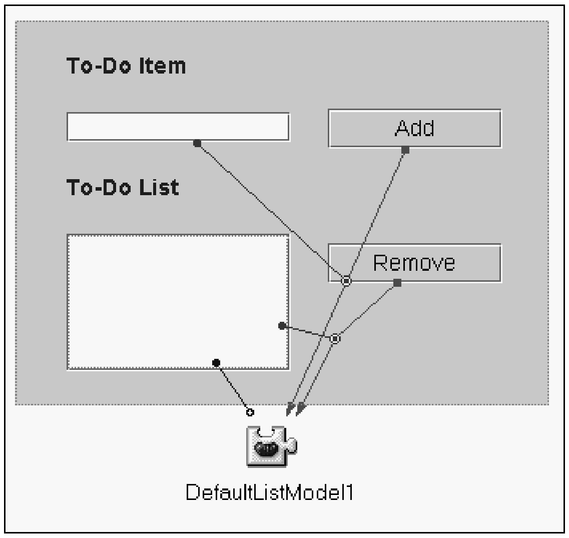
Click mouse button 2 on the DefaultListModel. Select **Connect** and then **this** from the pop-up window. Click mouse button 1 on the list and select **model** from the pop-up window. A blue line appears indicating a property-to-property connection.



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** and then **actionPerformed**.
2. Click mouse button 1 on the DefaultListModel, then select **Connectable Features** From the pop-up menu that appears, select **removeElementAt(int)**. Again, a dashed line appears, which means that more information is necessary. In this case, the value of the parameter for the *removeElementAt(int)* method is missing.
3. To make the parameter-from-property connection that specifies what to remove from the list, follow these steps:
   a. Move the mouse pointer on top of the dashed connection line.
   b. Click mouse button 2, then select **Connect**, then **index** from the pop-up menu that appears.
   c. Click mouse button 1 on the list, then select **selectedIndex** from the pop-up menu that appears.
   d. Save your work.

Your connections should now look like this:

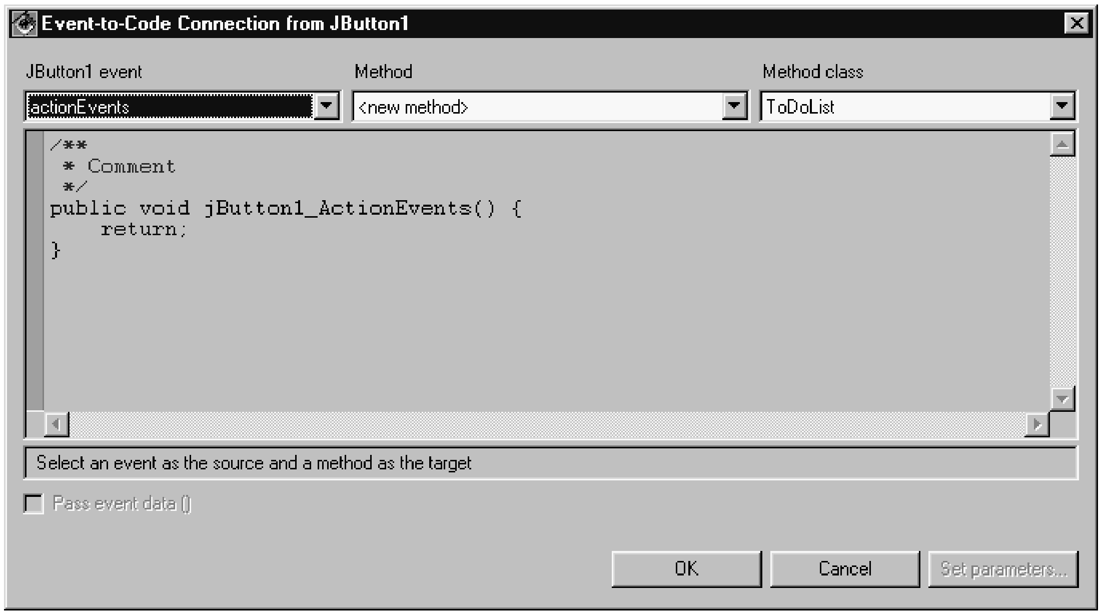


Event-to-Code Connection

Event-to-code programming allows you to associate a hand-written method with an event. In the first applet you created, you did not need to write any methods because VisualAge for Java generated them based on your visual elements and connections. However, sometimes additional logic is required. In this brief section you can link an event, like the click of a button, with your own hand-written methods and it is added to the Visual Composition Editor. An actual example of using this approach, however, is not shown here.

To link an event such as the click of a button with your own hand-written method, do the following:

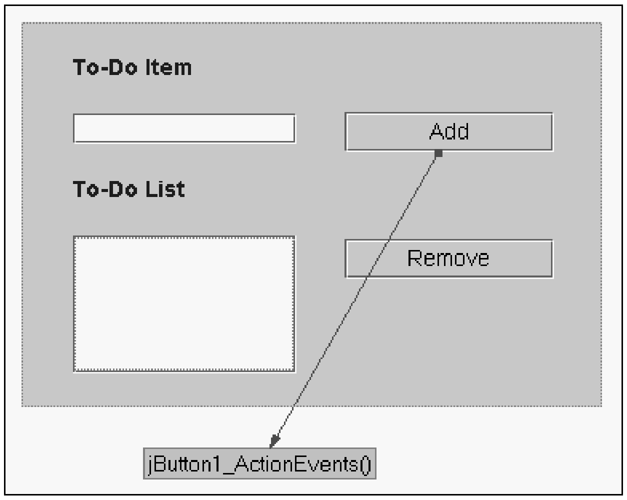
1. Click mouse button 2 on the **Add** button. Click mouse button 1 on **Event to Code** in the pop-up window. The Event-to-Code window appears.



```
/**
 * Comment
 */
public void jButton1_ActionEvents() {
    return;
}
```

Select an event as the source and a method as the target

2. At this point, add your code. Before the return statement, add a comment: `// testing event-to-code programming`. (You will get to add more real code using the Event-to-Code feature later.) Click **OK** to save it. An information message appears, telling you the text has been modified. Click **Yes**. Your method appears in the Visual Composition Editor as shown below.



3. If you had added a real method that performed a function then your method would be run when the **Add** button was clicked. Select the text box that contains the method name. Click mouse button 2 on the method. Select **Delete** from the pop-up window. Save your work.

For an introduction to some features of the VisualAge for Java IDE that help you write manual code quickly and neatly, go to Writing code by hand.

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

Saving and testing the To-Do List applet

You have already saved parts of your applet as you have been working. 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. It is good practice to 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 **Bean** 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 is used 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 **Run** from the tool bar

   The Applet Viewer starts with your applet in it.
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.

Note: As you design your applet, the Swing beans are presented with Sun's Metal look. For example, a JButton will look like a Metal JButton as opposed to a Windows JButton. You can add your own code to have your beans rendered in a different system's look and feel at runtime. For example, you could add code to have your beans have a Windows look and feel. Adding this user code is not covered in this document.

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 4. 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.

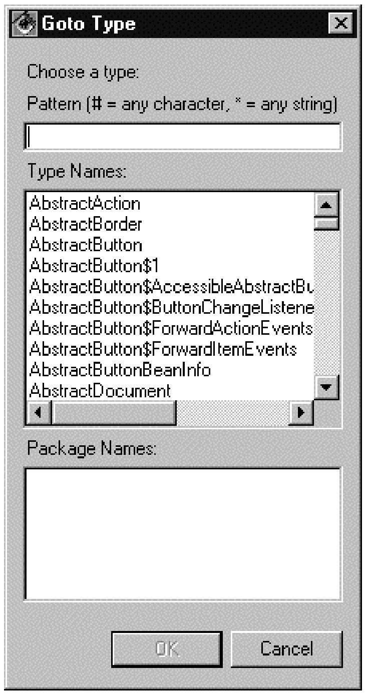
Before we update the applet to include state checking, we’ll go through the steps to find your applet and to create a versioned edition of it.

Note: This section assumes that you have completed the steps described in Building your first applet. 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. 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** and then **Type**. The GoTo Type secondary window appears:


Goto Type
Choose a type:
Pattern (# = any character, * = any string)
Type Names:
AbstractAction
AbstractBorder
AbstractButton
AbstractButton$1
AbstractButton$AccessibleAbstractBu
AbstractButton$ButtonChangeListene
AbstractButton$ForwardActionEvents
AbstractButton$ForwardItemEvents
AbstractButtonBeanInfo
AbstractDocument
Package Names:
OK
Cancel

3. Enter the name of your To-Do List applet (for example, *ToDoList*) in the **Pattern** field. As you enter the name, the **Type Names** list changes to include only the types (that is, the classes and interfaces) that match what you have entered so far.
4. Select the name of your To-Do List applet from the **Type 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 the versioned edition. If you decide you want to undo your 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 Managing editions of your code.

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

1. Ensure that the class for your applet is selected. From the **Selected** menu, select **Manage** and then **Version**. The Versioning Selected Items SmartGuide appears.
2. Ensure that **Automatic** is selected and click **OK**. 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 the working edition of the To-Do List applet that you just created.

[ENTERPRISE] Editions are managed by a team of developers in the Enterprise edition. Only owners of editions can version them. Also, versioning is often followed by releasing. For more information on the team development environment, see the online help or the *Getting Started* document for VisualAge for Java, Enterprise Edition.

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 no items are in the list. Here is how the **Remove** button should work:

- When the applet starts, the **Remove** button should be disabled.
- When an item is selected in the **To-Do List**, the **Remove** button should be enabled.
- When a selected item in the **To-Do List** has been deleted, the **Remove** button should be disabled.

Overview of adding the desired behavior to the Remove button

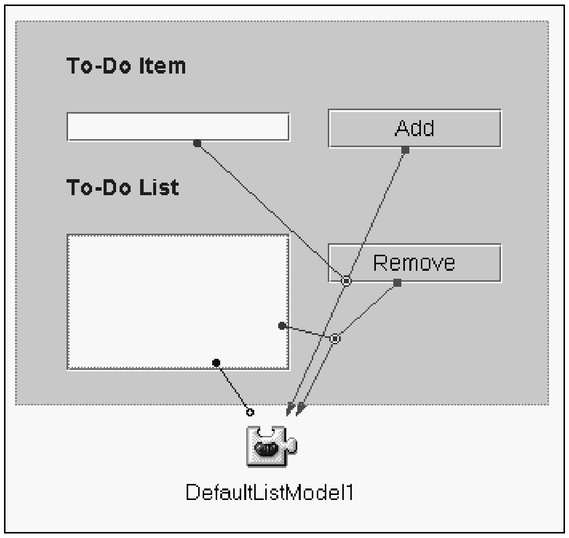
To get the desired behavior for the **Remove** button, you need to:

- 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 any item is selected in the **To-Do List**.
- Add an Event-to-Code connection to enable the button when an item is 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:

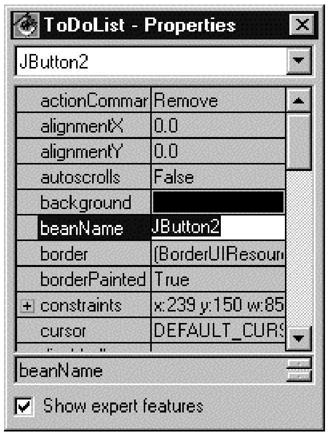
1. Select the class for your To-Do List applet in the Workbench.
2. From the **Selected** menu, select **OpenTo** and then **Visual Composition**.
3. The free-form surface appears. It should look like this:



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.



actionCommar	Remove
alignmentX	0.0
alignmentY	0.0
autoscrolls	False
background	
beanName	JButton2
border	(BorderUIResour
borderPainted	True
+ constraints	x:239 y:150 w:85
cursor	DEFAULT_CURS

2. Select the field to the right of **enabled**. Select **False** from the drop down list in this field and close the Properties 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 field beside **Types**:

```
boolean enableRemove(
```

Click **Types**. In the Types pop-up window, enter JL as the pattern. Select JList from the Type Names panel. Click **Insert**.

4. Add checkList as the parameter name for the JList type; then add a closing parenthesis. Your method should now be:

```
boolean enableRemove(com.sun.java.swing.JList
checkList)
```

This specifies a method that takes one parameter (a JList) and returns a boolean value.

5. Select **Finish** to generate the method.
6. Select the new *enableRemove(com.sun.java.swing.JList)* method from the **Methods** list and add the code to implement it. If you are viewing this document 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 boolean enableRemove(com.sun.java.swing.JList 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.

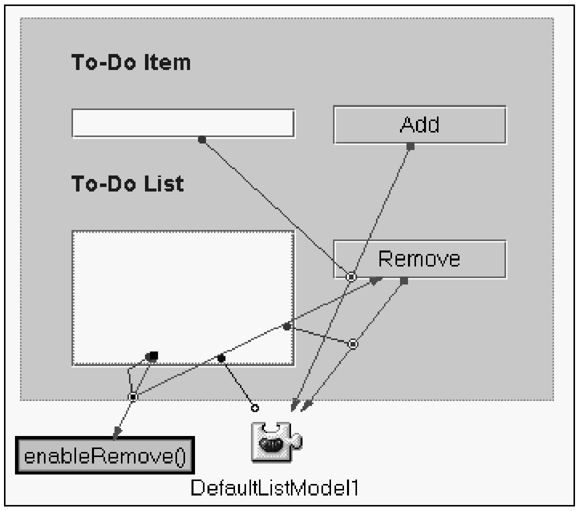
This simple method calls the *getSelectedIndex* method for its checkList parameter. If *getSelectedIndex* returns -1, there are no items selected in the list and *enableRemove(com.sun.java.swing.JList)* returns false (so the **Remove** button appears faded). Otherwise, *enableRemove(com.sun.java.swing.JList)* returns true (so the **Remove** button appears with solid black text like the **Add** button).

Add a connection to enable and disable 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 **listSelectionEvents** 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 free-form surface and selecting **Event to Code**. Select method **enableRemove()** and select **OK**.
4. The connection is incomplete because you need to provide information as to what JList should be checked for selected items. To complete the connection, select it, click mouse mouse button 2, and select **Connect**. Connect **checkList** to the JList1 *this* property.
5. The boolean that is returned from the **enableRemove** method is used to set the *enabled* property of the **Remove** button. Select the connection to the **enableRemove** method and click mouse button 2. Select **Connect** and select *normalResult*. Connect *normalResult* to the **Remove** button *enabled* property.

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



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 **Bean** menu.
2. To test the changes you made, select **Run** from the tool bar.
3. The Applet Viewer appears with your applet.
4. 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 nothing is selected in 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.

Chapter 5. Enhancing the To-Do List 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 To-Do lists to named files and open files containing To-Do lists.

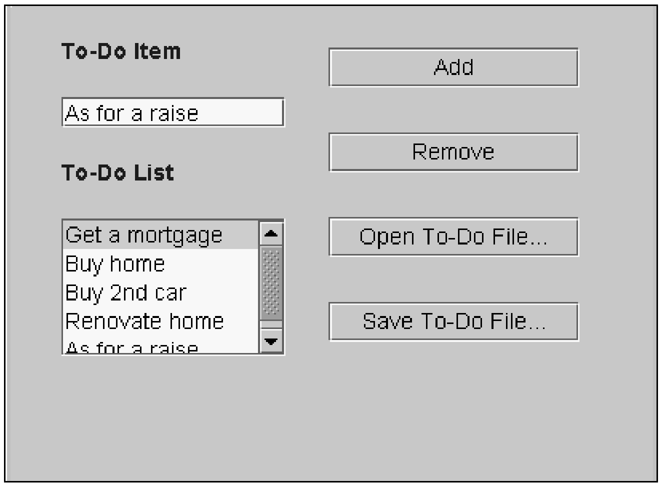
As you modify your applet, you will learn about:
- Creating new methods
- Adding business logic code in the Visual Composition Editor
- Updating the user interface
- Running code as an applet or an application

Note: This section assumes that you have completed the steps described in Adding state checking to your applet. 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 List program

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

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



Like your existing applet, the updated To-Do List program adds the text in the **To-Do Item** field to **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 File**, a file dialog appears 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 File**, a file dialog also appears. 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 List program. Because it needs access to the file system to read and write files, the To-Do List program must be run as an application rather than an applet. Java applets are not allowed to access the file system.

Steps for adding file access to the To-Do List program

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

1. Create new methods 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 JFileChooser beans for opening and saving files.
4. Add connections from the **Open To-Do File** >button.
5. Test the program to verify your work so far.
6. Add connections from the **Save To-Do File** >button.
7. Test the completed program.

The following sections describe these steps in detail.

Creating new methods

The next step in enhancing the To-Do List program is adding some methods that contain the logic for the To-Do List program to read and write To-Do files.

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

- Add a method for reading To-Do files.
- Add a method for writing To-Do files.

The following sections describe these tasks in more detail.

Creating a new method: adding a method for reading files

This section guides you through creating a method called *readToDoFile* that reads an input file.

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

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

Here are the detailed steps for creating this method:

1. Select the ToDoList class.
2. Select **Add** and then **Method** from the **Selected** menu. When the Create Method SmartGuide appears, enter the following method name: `void readToDoFile(File dirName, File fileName, DefaultListModel 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 DefaultListModel 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 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  (File dirName, File fileName,
        DefaultListModel fillList) {
        FileReader fileInStream = null;
        BufferedReader 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 FileReader(fileName);
           }
           catch (IOException e) {
                   System.err.println
                   ('IO exception opening To-Do File ' +fileName);
```

```
                        return;
                }
                dataInStream = new BufferedReader(fileInStream);
                // clear the existing entries from the list
                fillList.removeAllElements();
                try {
                        // for each line in the file create an item in the list
                        while ((result = dataInStream.readLine()) != null)  {
                                if (result.length() != 0)
                                        fillList.addElement(result);
                        }
                }
                catch (IOException e) {System.err.println
                      ('IO exception reading To-Do File ' +fileName);}
                try {
                        fileInStream.close();
                        dataInStream.close();
                }
                catch (IOException e) { System.err.println
                      ('IO exception closing To-Do File ' +fileName);}
             }
             else {
                     System.err.println
                        ('Null file name and/or directory reading To-Do File');
             }
        return;
    }
```

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 tests whether both the directory and the file name are non-null:

   ```
   FileReader fileInStream = null;
   BufferedReader 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 FileReader and to associate the FileReader with a BufferedReader. Using a BufferedReader makes it possible to read the file a line at a time.

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

3. Next, we clear the `fillList`. Then, a loop reads the file one 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.removeAllElements();
try {
        // for each line in the file create an item in the list
        while (((result = dataInStream.readLine()) != null) ) {
                if (result.length() != 0)
                        fillList.addElement(result);
        }
}
catch (IOException e) {System.err.println('IO exception reading To-Do File '
        +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 '
        +fileName);}
```

Creating a new method: adding a method for writing files

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

- Accept as arguments a directory, a file name, and a DefaultListModel object
- Write each item in the DefaultListModel object as a line in the file

Here are the detailed steps for creating this method:

1. Select the ToDoList class.
2. From the **Selected** menu, select **Add** and then **Method**. When the Create Method SmartGuide appears, enter the following in the method name: `void writeToDoFile(File dirName, File fileName, DefaultListModel fillList)`

   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 DefaultListModel object in the interface containing the items to be written to the file
3. Select **Finish** to generate the method.

4. Select the new *writeToDoFile* method and add the code to implement it. If you are viewing this document 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(File dirName, File fileName,
     DefaultListModel fillList) {
        FileWriter fileOutStream = null;
        PrintWriter dataOutStream;
        // carriage return and line feed constant
        String crlf = System.getProperties().getProperty('line.separator');
        // if valid directory and filenames passed, write the file from the list
        if ((dirName != null) && (fileName != null)) {
                try {
                        fileOutStream = new FileWriter(fileName);
                }
                catch (IOException e) {
                        System.err.println
                        ('IO exception opening To-Do File ' +fileName);
                        return;
                }
                dataOutStream = new PrintWriter(fileOutStream);
                // for every item in the list, write a line to the output file
                for (int i = 0; i < fillList.getSize(); i++) {
                        try {
                                dataOutStream.write(fillList.get(i)+crlf);
                        }
                        catch (Exception e) { System.err.println
                        ('Exception writing To-Do File '  +fileName);}
                }
                try {
                        fileOutStream.close();
                        dataOutStream.close();
                }
                catch (IOException e) { System.err.println
                ('IO exception closing To-Do File ' +fileName);}
        }
        else {
                System.err.println
                ('Null file name and/or directory writing To-Do File');
        }
        return;
}
```

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.getSize(); i++) {
        try {
                dataOutStream.write(fillList.get(i)+crlf);
        }
        catch (Exception e) { System.err.println('Exception writing To-Do File '
                +fileName);}
}
```

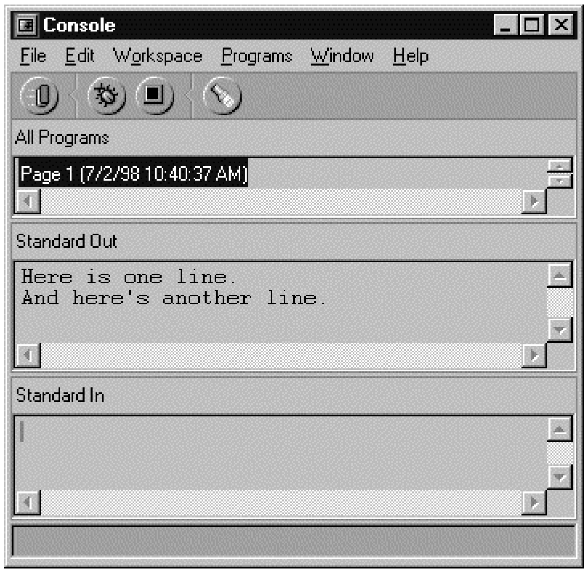
This loop goes through each item in `fillList`. Each item is appended with `crlf` (a String consisting of the line separator characters) and written to the file. The line separator characters force each item to be written on a separate line in the file.

Using the Scrapbook to test code

Before continuing, let’s pause and consider the line separator for a moment. Suppose 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:

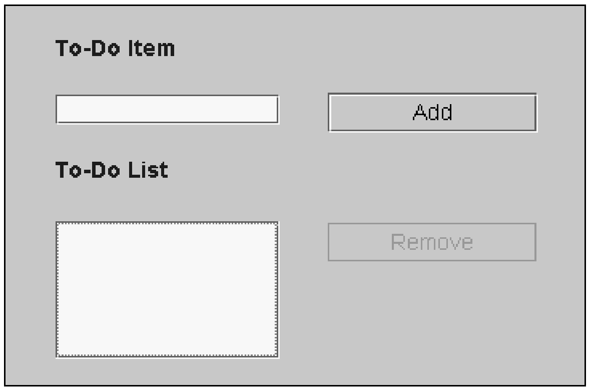

Console
File Edit Workspace Programs Window Help
All Programs
Page 1 (7/2/98 10:40:37 AM)
Standard Out
Standard In

```
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 of the steps that added logic to the ToDoList 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:

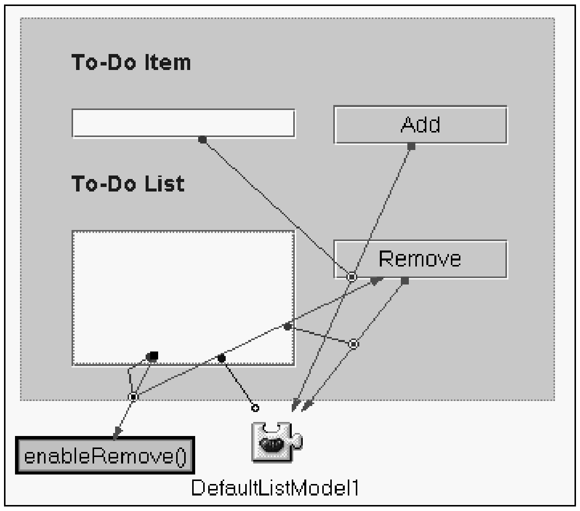


You need to add two new buttons to this user interface:

- An **Open To-Do File** button to trigger opening a file to read into the To-Do list
- A **Save To-Do File** button to trigger saving the contents of the To-Do list to a file

To add these two buttons:

1. Select the ToDoList class for your To-Do List applet.
2. From the **Selected** menu, select **Open To** and then **Visual Composition**.
3. The free-form surface appears. It should look like this:



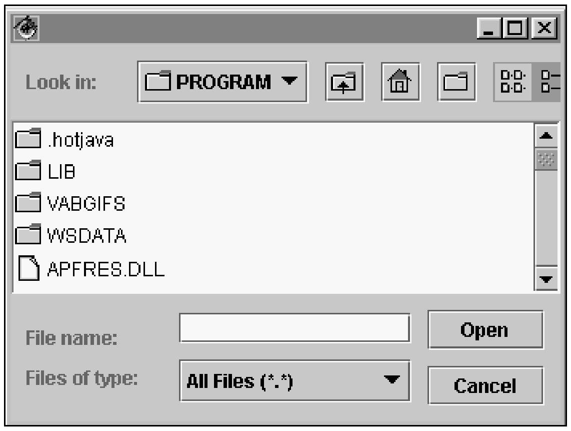
4. Select a JButton bean and add a button under the existing **Remove** button. You may need to move your **Add** and **Remove** buttons or lengthen the free-form surface to make space for your new button.
5. Select the button you just added and change its text to **Open To-Do File...**. To change the text:
   - Open **Properties** for the new JButton (JButton3).
   - Change the **text** value to *Open To-Do File....*
6. Follow the same procedure to add another button below JButton3. 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 **Add** button. Hold down the Ctrl key and select the **Remove**, **Open To-Do File** and **Save To-Do File** buttons so that all four buttons are selected. The **Save To-Do File** button, the last bean selected, has solid selection handles, indicating that it is the *anchor bean*. The anchor bean is the bean that acts as the guide for resizing or the bean that the other selected beans match.
   - 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 and select the **Open To-Do File**, **Remove**, and **Add** buttons so that all four buttons are selected. The **Add** button becomes the anchor bean.
   - Select **Align Left** from the tool bar.
9. Distribute evenly all four buttons:
   - Because you have all buttons already selected, click **Distribute Vertically** on the tool bar.

You have added the two new buttons for the To-Do List program. Now you are ready to associate them with some action.

Adding JFileChooser beans to the free-form surface

Now that you have added the new buttons, the next step is to add file dialog beans for opening files and saving files. Later, you will use these file dialog beans to obtain a file selection and provide parameters to the **readToDoFile** and **writeToDoFile** methods.

These file dialog beans are preview beans for Swing that are not part of the Java Foundation Classes. Note that the Java Foundation Classes (JFC) present these file dialogs in a platform-independent representation as shown below:



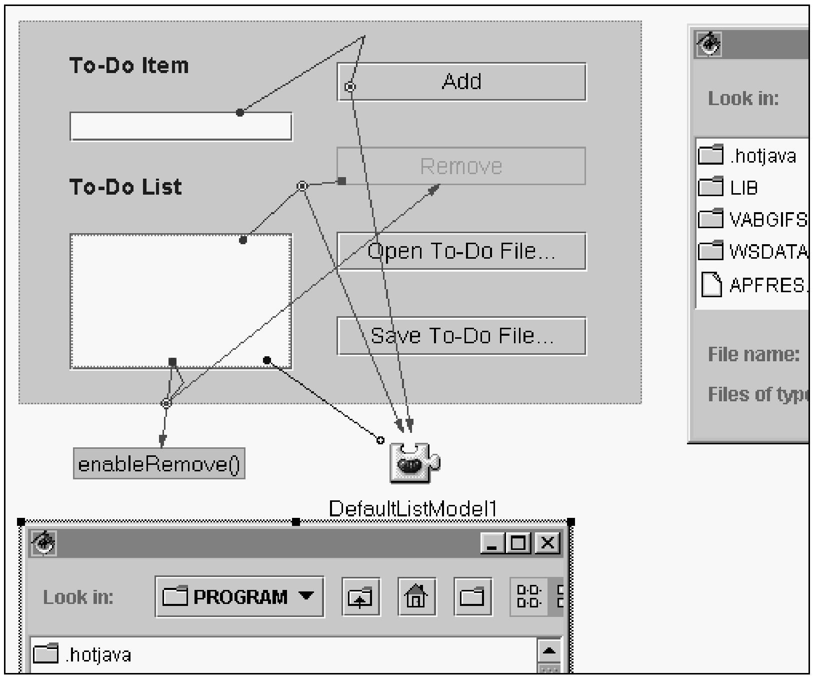
In the finished To-Do List program, a file dialog appears when a user selects the **Open To-Do File** or **Save To-Do File** button. In the file dialogs, the user specifies the name of the file to open or save.

To add the file dialog beans:

1. Select the **JFrame** bean from the palette.
2. Add the JFrame bean to the right side of the applet area, that is, outside of the gray area. Add a second JFrame bean below JFrame1. Select both of them and open the Property window to change the layout to BorderLayout.
3. Select **Choose Bean** from the palette.

4. When the Choose Bean window appears, select **Browse** to specify the *com.sun.java.swing.preview.JFileChooser* class. Click **OK**. Place the JFileChooser bean (which is the file dialog itself) inside JFrame1.
5. Add a second JFileChooser dialog bean inside JFrame2.

Your free-form surface should look similar to the one below.



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

Note: The exact positions of 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 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.

The To-Do List applet should perform the following actions when a user selects the **Open To-Do File** button:

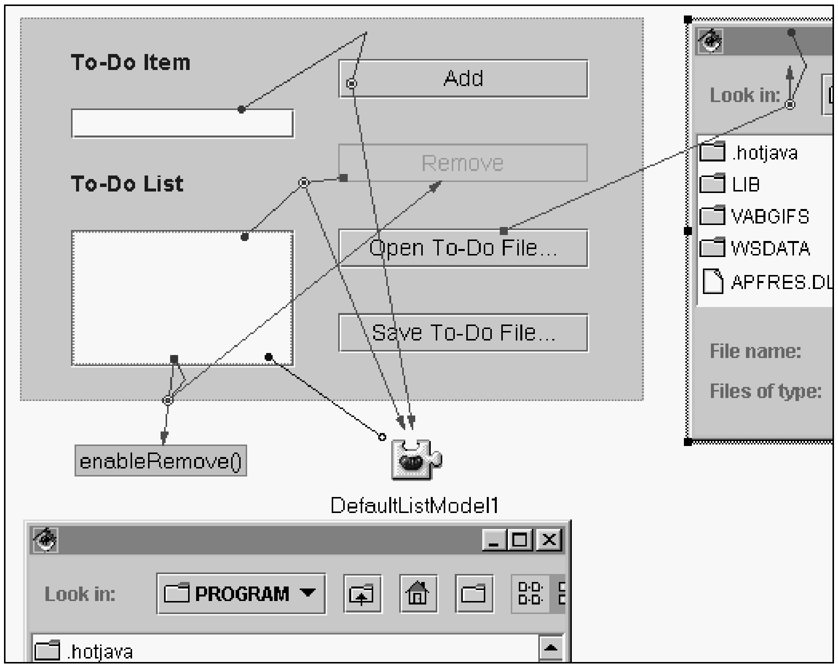
1. Show the file dialog.
2. Dispose of the file dialog.
3. Invoke the *readToDoFile* method.

You implement actions 1 and 2 by making connections between the **Open To-Do File** button and the JFileChooser bean for opening files. You implement action 3 by making a connection between the **Open To-Do File** button and the DefaultListModel bean.

Create the connection to show and dispose of the open file dialog

1. Select the **Open To-Do File** button and click mouse button 2. Select **Connect** and then **actionPerformed** 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 JFileChooser bean to the right of the applet. From the pop-up menu that appears, select **Connectable Features**. Choose **Methods** from the End connection to window. Select **showOpenDialog(java.awt.Component)**. Click **OK**.
3. Select the connection just completed and click mouse button 2. Select **Connect** and then **parent** from the pop-up menu. Click the frame containing the file chooser and select **this** from the pop-up menu.

Now the free-from surface should look like this:



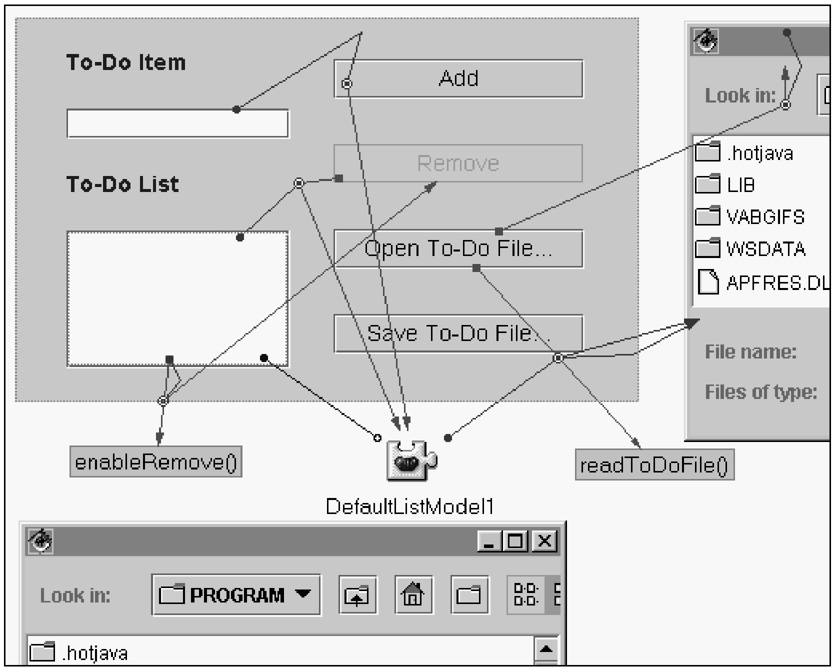
You have completed all the connections between the **Open To-Do File** button and the JFileChooser that get data from a file and put it in your To-Do list. Now you are ready to make the connection that invokes *readToDoFile* in the DefaultListModel bean.

Create the connection to invoke readToDoFile

1. Select the **Open To-Do File** button and click mouse button 2. Select **Connect** and then **actionPerformed** from the pop-up menu that appears.
2. Click mouse button 1 on the free-form surface and select **Event to Code**.
3. In the Event to Code window, select method **readToDoFile(java.io.File, java.io.File, com.sun.java.swing.DefaultListModel)** and then select **OK**. The connection that appears is incomplete because *readToDoFile* takes three parameters: a directory name, a file name, and a DefaultListModel object. Begin by specifying the directory name:
   - Select the connection and click mouse button 2.
   - Select **Connect** and 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.
   - Move the mouse pointer to JFileChooser1 and click mouse button 1. Select **Connectable Features** from the pop-up menu that appears.
   - Choose **Method** from the End connection to secondary window, select **getCurrentDirectory()** and 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 readToDoFile method; click mouse button 2.
   - Select **Connect** and then **fileName** from the pop-up menu that appears.
   - Click mouse button 1 on JFileChooser1. Select **Connectable Features** from the pop-up menu that appears.
   - Choose **Method** from the secondary window, select **getSelectedFile()** and 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 readToDoFile method; click mouse button 2.
   - Select **Connect** and then **fillList** from the pop-up menu that appears.
   - Click mouse button 1 on the DefaultListModel bean and select **this** from the pop-up menu that appears. This last connection is significant. It specifies that the DefaultListModel bean in the user interface is the fillList parameter for *readToDoFile*. In other words, the DefaultListModel

bean in the interface is the DefaultListModel 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 List program.

Testing the Open To-Do File button

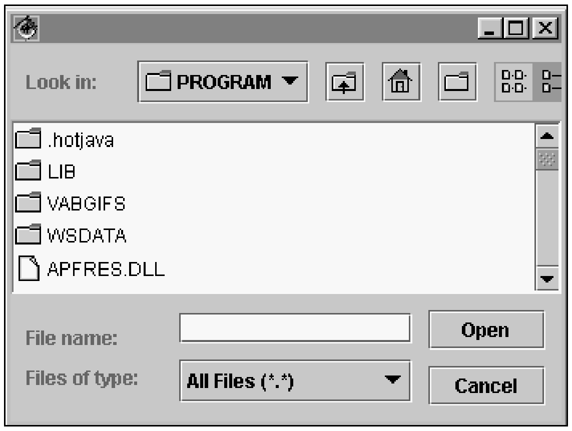
Now that you have made all the connections for the **Open To-Do File** button, you are ready to test the work you have done so far.

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

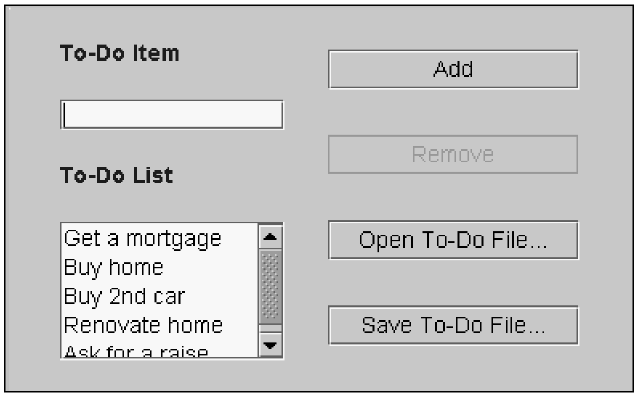
1. First, prepare a simple text file to use for testing. Use the Scrapbook window to create and save a sample To-Do file called `test1.txt` with the following lines in it:

```
Get a mortgage
Buy home
Buy 2nd car
Renovate home
Ask for a raise
```

2. Save your current work in the Visual Composition Editor by selecting **Save Bean** from the **Bean** menu. VisualAge for Java generates code to implement the connections you specified in the last step.
3. Select **Run** from the tool bar.
4. The To-Do List program appears.
5. Select the **Open To-Do File** button. A file dialog like the one below should appear. Note that the bottom right-hand button says 'Open'. If you simply leave the cursor on Open without clicking, hover help tells you what the button will do.



6. From this file dialog, go to the drive and directory in which the `test1.txt` file is located. To select a drive, click **Program** and select a drive. The directories and files from the drive are shown. Select your file and click **Open**.
7. The **To-Do List** in your program should now be loaded with the items from the `test1.txt` file:



Now that you have tested your current progress on the To-Do List 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.

The To-Do File program should perform the following actions 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 to write the list of items in the To-Do list to a file that is selected in the file dialog.

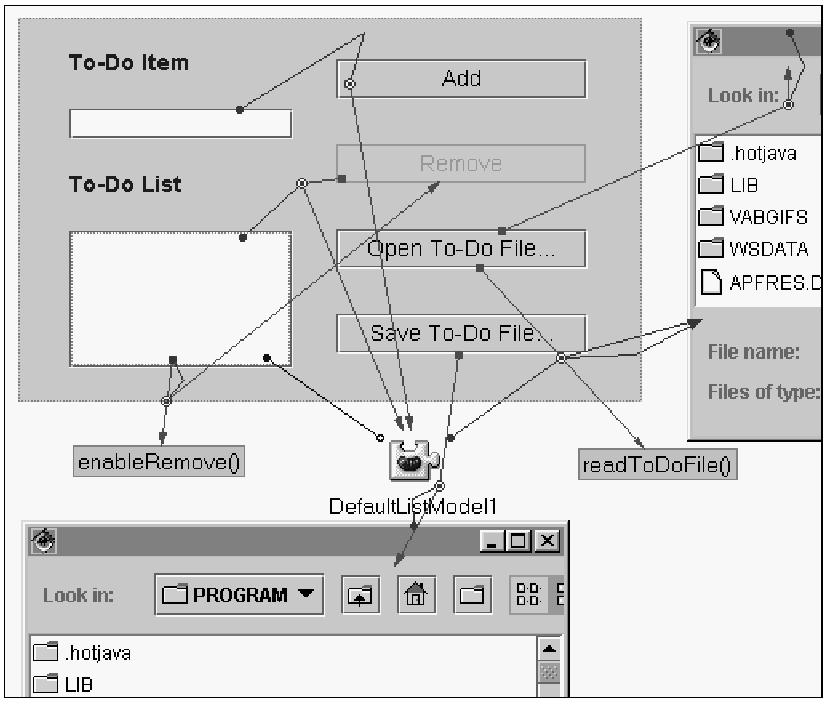
You implement actions 1 and 2 by making connections between the **Save To-Do File** button and the Save JFileChooser dialog bean. You have already implemented action 3 by making the connection between the Save To-Do File button and the readToDoFile method.

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

Create the connection to show and dispose of the save file dialog

1. Select the **Save To-Do File** button and click mouse button 2. Select **Connect** and then **actionPerformed** from the pop-up menu that appears.
2. Click mouse button 1 on JFileChooser2. From the pop-up menu that appears, select **Connectable Features**. Choose **Method** from the End connection to window. Select **showSaveDialog(java.awt.Component)**. Click **OK**.
3. Select the connection just completed and click mouse button 2. Select **Connect** and then **parent** from the pop-up menu. Click the frame containing the file chooser and select **this** from the pop-up menu.

Now the free-form surface should look like this:

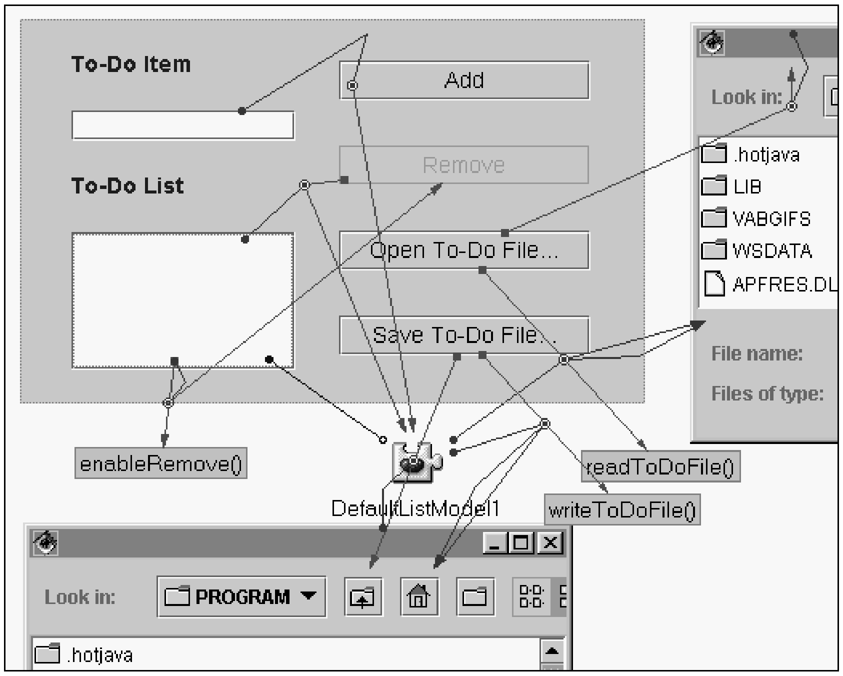


Now you are ready to make the connection that invokes *writeToDoFile*.

Create the connection to invoke writeToDoFile

1. Select the **Save To-Do File** button and click mouse button 2. Select **Connect** and then **actionPerformed** from the pop-up menu that appears.
2. Click mouse button 2 on the the free-form surface. From the pop-up menu that appears, select Event to Code.
3. In the Event to Code window, select **writeToDoFile(java.io.File, java.io.File, com.sun.java.swing.DefaultList Model)**. The connection that appears is incomplete because *writeToDoFile* takes three parameters: a directory name, a file name, and a DefaultListModel object. Begin by specifying the directory name:
   - Select the connection and click mouse button 2.
   - Select **Connect** and 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 ToDoList class.
   - Move the mouse pointer to FileChooser2 and click mouse button 1. Select **Connectable Features** from the pop-up menu that appears.
   - Choose **Method** from the End connection to secondary window. Select **getCurrentDirectory()**, and then select **OK.**

4. To specify the file name:
   - Select the connection between the **Save To-Do File** button and the writeToDoFile method and click mouse button 2.
   - Select **Connect** and then **fileName** from the pop-up menu that appears.
   - Click mouse button 1 on FileChooser2. Select **Connectable Features** from the pop-up menu that appears.
   - Choose **Method** from the secondary window, select **getSelectedFile()** and then select **OK**.
5. There is still one parameter required before the connection is complete. To specify the DefaultListModel object:
   - Select the connection between the **Save To-Do File** button and the writeToDoFile method; click mouse button 2.
   - Select **Connect** then **fillList** from the pop-up menu that appears.
   - Click mouse button 1 on the DefaultListModel 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.

Saving and testing the completed To-Do List program

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

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

1. Select **Save Bean** from the **Bean** 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. Select **Run** from the tool bar.
3. 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**:
     - *Get paint*
     - *Get wallpaper*
     - *Spouse says OK?*
     - *Start painting*
     - *Start wallpapering*
   - Select **Save To-Do File.** A save file dialog should appear. In this dialog, go to the directory in which you saved the `test1.txt` file for testing the Open To-Do File button. Enter the file name `test2.txt` and select **Save**.
4. 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**.
5. 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 the ToDoList class.

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

Chapter 6. What Else Can You Do With the Visual Composition Editor?

In Building your first applet, 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 skills, 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

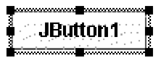
After you add beans to an applet, you will often want to align them, size them, or perform similar tasks. Before you can align or size your beans, however, you must learn to manipulate them. This section introduces you to the following tasks:

- 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.

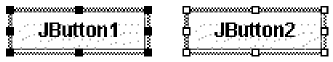


Note: Beans that cannot be sized do not have selection handles. Instead, these beans change their background color when they are selected. Beans with this behavior include nonvisual beans and menu beans.

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 status 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 is the *anchor* bean. The other selected beans have hollow selection handles.



The anchor bean is important when performing operations such as bean sizing and alignment. The other selected beans set their position or size to the position or size of the anchor bean. You can change the anchor bean by holding the shift key and selecting the bean you want to be the anchor.

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 the number of beans selected (for example, `3 beans selected`).

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 at which 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. 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, hold the Ctrl key, 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 before 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: Beans that are containers (such as a JApplet or a Frame) have a layout property. This property provides specific layout managers that control the positioning of beans within the container. Using a layout manager is the preferred way to create a user interface. However, if you use a <null> layout, the Visual Composition Editor provides tools for aligning and positioning beans.

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 in only one direction, 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.

Aligning beans

To align beans with other beans in a <null> layout, follow these steps:

1. Select the beans you want to align, ensuring that 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 that 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** and 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 that uses <null> layout, 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** and then **Distribute.** 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 **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

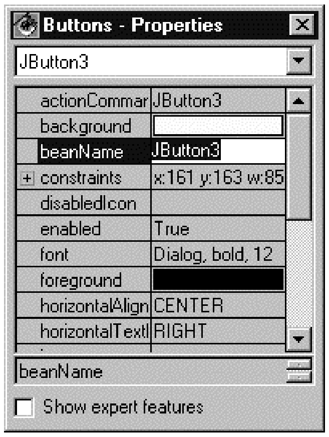
To open the Properties window for a bean, do any of the following:

- 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:



Buttons - Properties

JButton3

actionCommar	JButton3
background	⬜
beanName	JButton3
+ constraints	x:161 y:163 w:85
disabledIcon	
enabled	True
font	Dialog, bold, 12
foreground	⬛
horizontalAlign	CENTER
horizontalTextI	RIGHT

beanName

☐ 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 JTextField 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 (for example, color and font) can be changed through a second window displayed for that purpose.

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

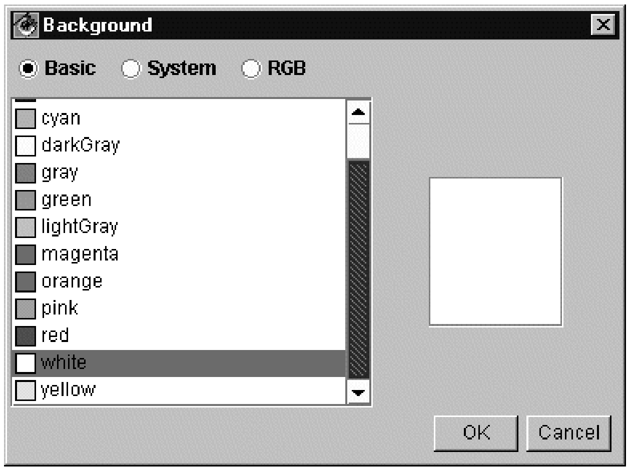
Another enhancement that you can make to your visual beans is to change the colors and fonts that the beans use.

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.

Changing the color of a bean

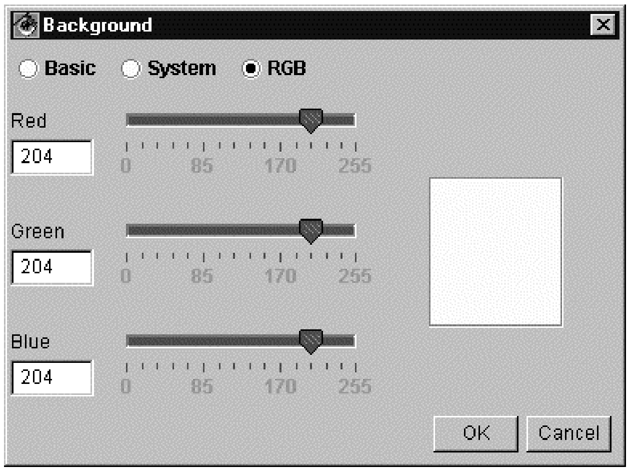
1. In the Visual Composition Editor, double-click on a JButton bean whose color you want to change. The Properties window appears.
2. To change the background color of a bean, select the value for the **background** property in the Property window. Select the button that appears: ...

   The Backgound window opens:



3. In the Background window, click mouse button 1 on the color you want to use. The color appears in the color pane. Then select **OK**.

Now double-click on a JLabel bean. Follow the same steps, again choosing the **background** property. This time a slightly different window appears, as shown below. To alter colors, use the slider controls, or select Basic or System and choose a color.

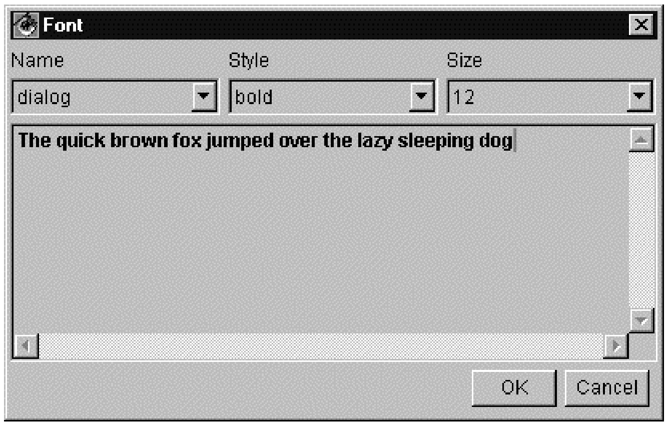


Note: You cannot change the color of beans in Menus.

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 Font window opens:



Name: dialog
Style: bold
Size: 12

The quick brown fox jumped over the lazy sleeping dog

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

Note: Some beans, such as Menu beans, 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 basic colors in the 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 Building your first applet, 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, 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 both source and target events are specified in the connection’s Property window, the two values stay in sync.

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

Event-to-code
Event-to-code connections run some code when an event occurs.

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

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

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

Event-to-code and parameter-from-code connections enable you to connect to 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.

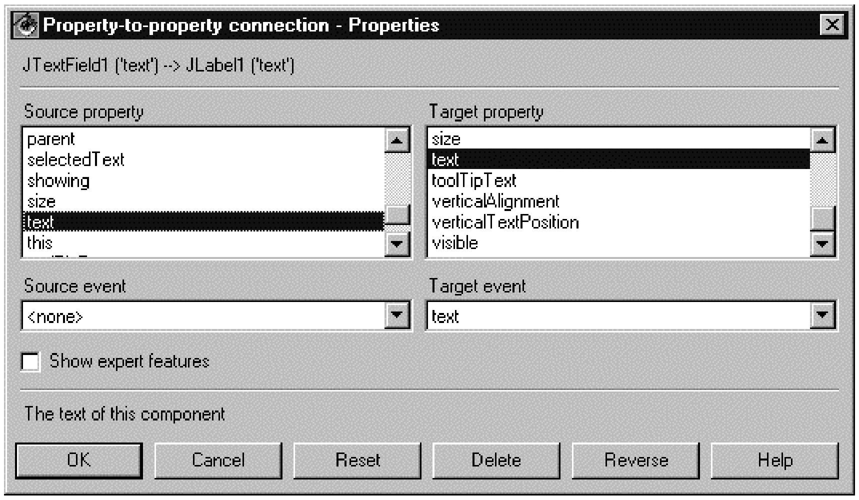
Note: If a particular bean method, property, or event does not appear in the bean's preferred feature list in its bean or connection pop-up menu, you can select **Connect** and then**Connectable Features** from the bean pop-up or connection pop-up menu to display a complete list. The list of methods,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

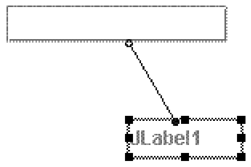
1. Create a new applet using the Create Applet SmartGuide:
   - Select **Create Applet** from the Workbench tool bar.
   - In the Create Applet SmartGuide, enter a name for the applet and specify a project and package for the applet. For **Superclass**, use **Browse** and enter JApplet as the pattern to get com.sun.java.swing.JApplet. Ensure **Compose the class visually** is selected and select **Finish**.
2. When the Visual Composition Editor opens on your new applet, place a JTextField bean and a JLabel bean within the default Applet bean.
3. Connect the *text* property of the JTextField bean to the *text* property of the JLabel bean:
   - Select the JTextField bean and click mouse button 2. Select **Connect** and then **text** from the pop-up menu that appears.
   - Click mouse button 1 on the JLabel 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.



5. In the Properties window, select **text** 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 JLabel text becomes JTextField1.

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 feature.

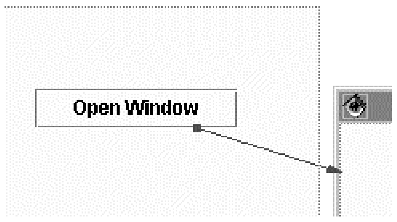
Event-to-method connections

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. A simple example of an event-to-method connection follows:

1. Place a JButton bean within the default JApplet bean in the Visual Composition Editor. Change the text of this button to *Open Window*.
2. Place a JFrame bean on the free-form surface of the Visual Composition Editor.
3. Connect the *actionPerformed* event of the JButton to the *show* method of the JFrame bean. To do this connection, click the title bar of JFrame (not inside the box itself). The frame appears 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:

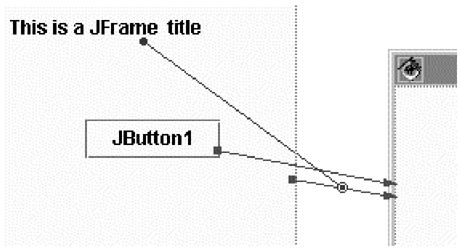
Connecting an event to a property

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 JLabel 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 JFrame title.*
2. Place a JButton bean within the applet bean in the Visual Composition Editor.
3. Place a JFrame bean on the free-form surface of the Visual Composition Editor.
4. Connect the *actionPerformed* event of the JButton to the *show* method of the JFrame bean. Remember to click the title bar of JFrame to see the *show* method.
5. Connect the *componentShown* event of the applet bean to the *title* property of the JFrame bean.

6. Now, to provide the parameter for the event-to-property connection you just created, connect the *text* property of the JLabel 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 is set to *This is a JFrame title*. This example is rather contrived, but it conveys the idea. For more information, see Connection parameters.

Event-to-code connections

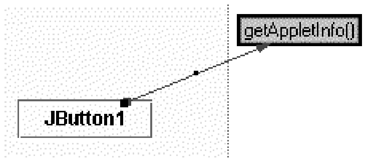
Event-to-code connections run a given method when a certain event takes place. This provides a way to implement or alter applet behavior directly through the use of the Java language. The target of an event-to-code method can be any method in the class that you are manipulating in the Visual Composition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

Note: An event-to-method connection is made between two beans. An event-to-code 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-code connection is green. To create an event-to-code connection:

1. Select the source bean (for example, a JButton). Click mouse button 2 to display the bean's pop-up menu. Select **Connect** and then select an event, such as *actionPerformed*. 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 Code** from the pop-up menu.
3. The resulting window allows you to pick from the list of available methods, or to create a new method.
4. Once you've selected a method, select **OK** to complete the connection.

The connection is drawn between the source bean and a movable text box that contains the name of the method.



Parameter connections

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

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

The color for a parameter connection line is purple.

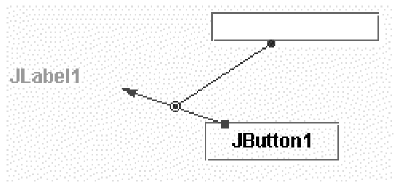
Parameter-from-property

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. You then select the appropriate property 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 JTextField bean's *text* property to the connection between the JButton bean and the DefaultListModel bean, we were making a parameter-from-property connection. Refer to Connecting beans 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 JLabel bean, a JTextField bean, and a JButton bean within the applet bean in the Visual Composition Editor.
2. Connect the *actionPerformed(awt.java.event.ActionEvent)* event of the JButton bean to the *text* property of the JLabel bean.
3. Now, make the parameter-from-property connection by connecting the *text* property of the JTextField 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-code

Parameter-from-code 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 *stringFromCode* in the applet class that returns the text *this is a string*. An example of a parameter-from-code connection follows. Unlike other connection types, a parameter-from-code connection is initiated from the parameter name in the connection's pop-up menu and is terminated as follows:

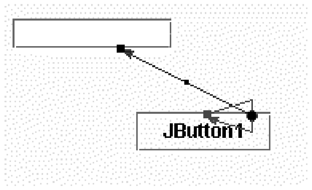
1. Place a JLabel bean and a JButton bean within the default JApplet bean in the Visual Composition Editor.
2. Connect the *actionPerformed* event of the JButton to the *text* property of the JLabel bean.
3. Now, create the parameter-from-code connection by connecting the *value* property in the connection pop-up menu to the method as follows:
   - Click mouse button 2 on the connection; select **Connect** and 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 Code** from the pop-up menu.
   - The resulting window allows you to pick from the list of available methods. In our case, the method *stringFromCode()* appears (this is the method that we created in the applet class).
   - Once you've selected a method, select **OK** to complete the 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 JButton bean and a JTextField bean within the default applet bean in the Visual Composition Editor.
2. Connect the *actionPerformed* event of the JButton bean to the *text* property of the JButton 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 JTextField 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.

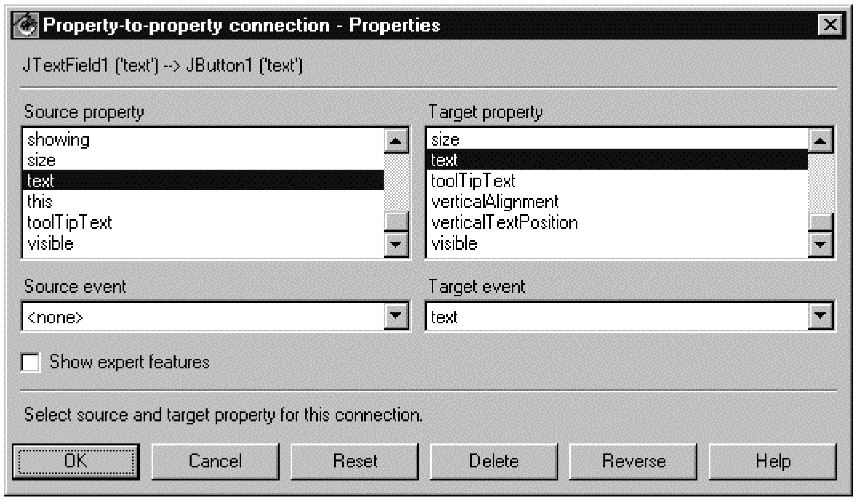
When you test the applet, type a string into the text field and then select the Button. The string becomes the label for the Button.



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

JTextField1 ('text') --> JButton1 ('text')

Source property: showing, size, **text** (selected), this, toolTipText, visible

Target property: size, **text** (selected), toolTipText, verticalAlignment, verticalTextPosition, visible

Source event: <none>

Target event: text

☐ 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.

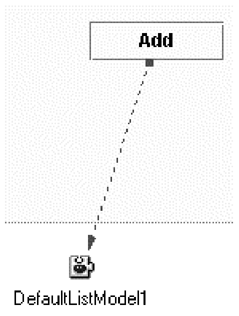
When you have finished changing the connection’s properties, select **OK**.

Connection parameters

Event-to-method and event-to-code 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 applet, you connected the JButton bean’s *actionPerformed* event to the DefaultListModel bean’s *addElement(java.langObject)* 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 *addElement(java.lang.Object)* method takes one parameter, an Object. A method named *insert ElementAt(java.lang.Object, int)* takes two parameters, an Object and an int.

When you have specified all of 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.

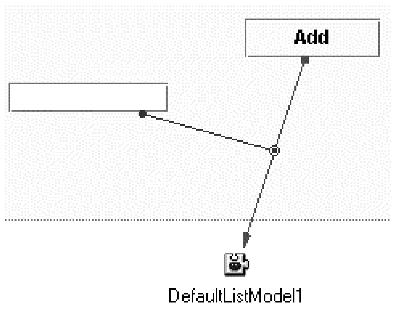
Properties as parameters

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 JTextField bean is used as the parameter of the event-to-method connection between the **Add** Button and the DefaultListModel bean.



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

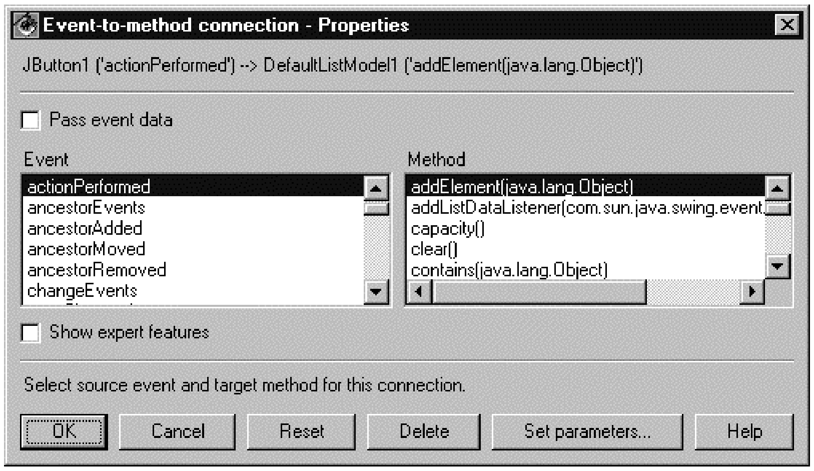
Constants as parameters

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:

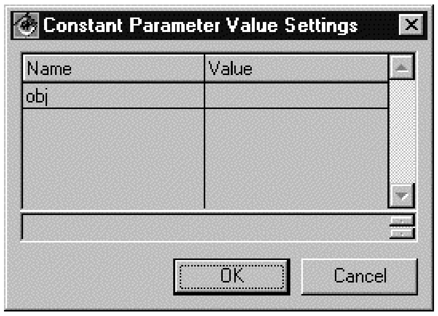
1. Double-click on the event-to-method connection.

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



Event-to-method connection - Properties
JButton1 ('actionPerformed') --> DefaultListModel1 ('addElement(java.lang.Object)')
Pass event data
Event
actionPerformed
ancestorEvents
ancestorAdded
ancestorMoved
ancestorRemoved
changeEvents
Method
addElement(java.lang.Object)
addListDataListener(com.sun.java.swing.event
capacity()
clear()
contains(java.lang.Object)
Show expert features
Select source event and target method for this connection.
OK
Cancel
Reset
Delete
Set parameters...
Help

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



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.

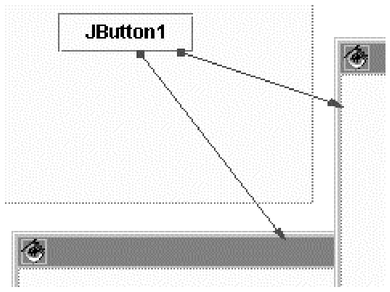
Deleting connections

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

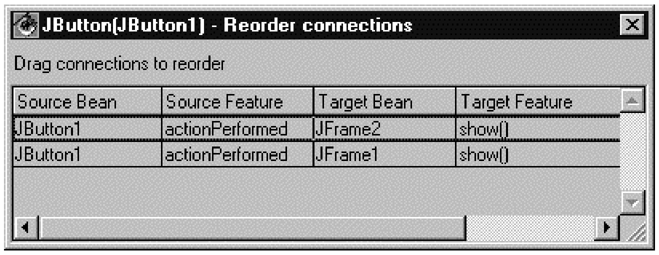
When you make several connections from the same event or property of a bean, the connections run in the order in which they were made. However, if you create the connections in a different order than the order in which you want to run them, you can reorder them. Add a JButton and two JFrame containers. Connect an *actionPerformed* event from the button to a *show* method at one frame. Make the same connection to the second frame. You now have two connections to open frames. When you run the applet, one frame will appear followed by the second after you click the button.



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:



JButton(JButton1) - Reorder connections

Drag connections to reorder

Source Bean	Source Feature	Target Bean	Target Feature
JButton1	actionPerformed	JFrame2	show()
JButton1	actionPerformed	JFrame1	show()

The Reorder connections window contains all of the connections for the bean you selected. In this example, the *(JButton1, actionPerformed* **->** *JFrame2, show()* connection is the first one that runs.

2. In the Reorder connections window use the appropriate mouse button to reverse the frame appearance order. 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. Rerun your applet to see the effect.

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.

You can 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 that 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-code 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 move 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.

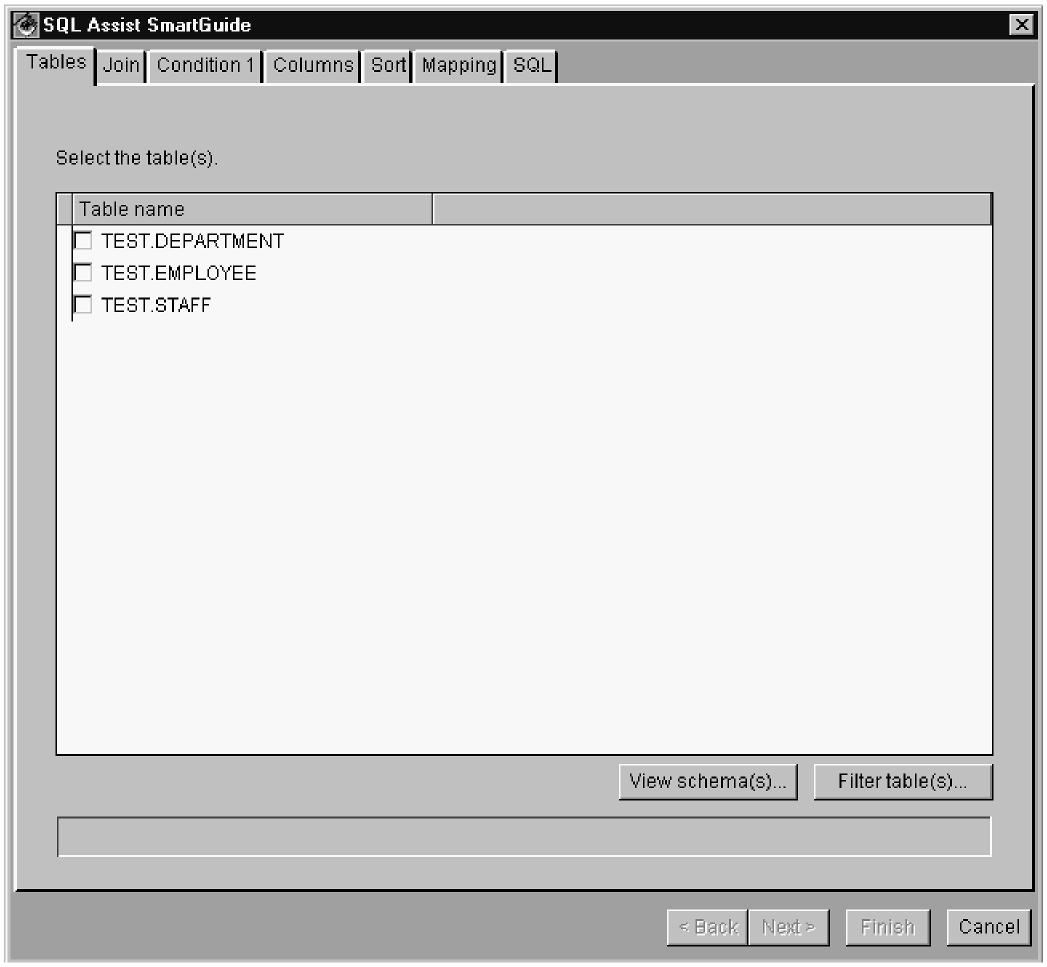
Working with relational data: the Select bean

VisualAge for Java supports access to relational databases through JDBC. You can access relational data in an applet or application by using the Data Access beans on the Visual Composition Editor beans palette. (Before you can use the Data Access beans, you must first use the Quick Start window to add the Data Access beans feature to Visual Age for Java, and you must change the classpath. For more information on accessing relational data, refer to the online help.)

The Data Access beans comprise a Select bean and a DBNavigator bean. The Select bean gives you a fast, easy to use, visual programming way of accessing relational data in your applet or application.

To access relational data using the Select bean:

1. Add the Select bean to the Visual Composition Editor surface.
2. Edit the Select bean properties. The query property allows you to define the following things:
   - **Connection alias**. This identifies the characteristics of the database connection for the Select bean. These include characteristics such as the URL for the connection, and the user ID and password to be passed with the connection request.
   - **SQL specification**. This specifies the SQL statement for the Select bean. You can use the SQL editor that is provided to enter the SQL statement manually, or you can use the SQL Assist SmartGuide to compose the SQL statement visually. You can select one or more tables, join tables, specify search conditions, identify columns for display, sort the results, map the data types of the result columns into Java classes, and view the resulting SQL statement. You can even do a test run of the query.



Select the table(s).

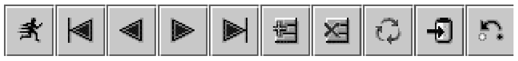
Table name
☐ TEST.DEPARTMENT
☐ TEST.EMPLOYEE
☐ TEST.STAFF

3. Connect the Select bean to a visual component of your applet or application, such as a button bean. When a user uses the application or applet, and selects the visual component, for example clicks the button, the SQL statement for the Select bean retrieves a result set.

You can also use the Select bean to apply changes that users make to a result row, even deletion of a row, and commit the changes to the database. In these cases, you use update and delete methods provided by the Select bean.

Adding buttons for relational database access: the DBNavigator bean

The DBNavigator bean gives you an easy way of adding to your program element a set of buttons that navigate the rows of a result set and perform various relational database operations.



For example, one of the buttons makes the next row in the result set the current row, and another button commits changes to the database.

The DBNavigator bean is used in conjunction with a Select bean. As described above, the Select bean is used to retrieve a result set from a relational database. The DBNavigator bean operates on the result set.

To use the DBNavigator bean:

1. Add the DBNavigator bean to the Visual Composition Editor surface.
2. Edit DBNavigator bean properties. Among the properties that you edit are properties that specify which buttons will be displayed.
3. Connect the DBNavigator bean to a Select bean.

Chapter 7. 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 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

[ENTERPRISE] Editions are managed by a team of developers. Packages and projects can only be versioned by their owners; classes can only be versioned by their developers. Versioning an edition is often followed by releasing. For information on the team development environment, see *Getting Started* for VisualAge for Java, Enterprise Edition.

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.

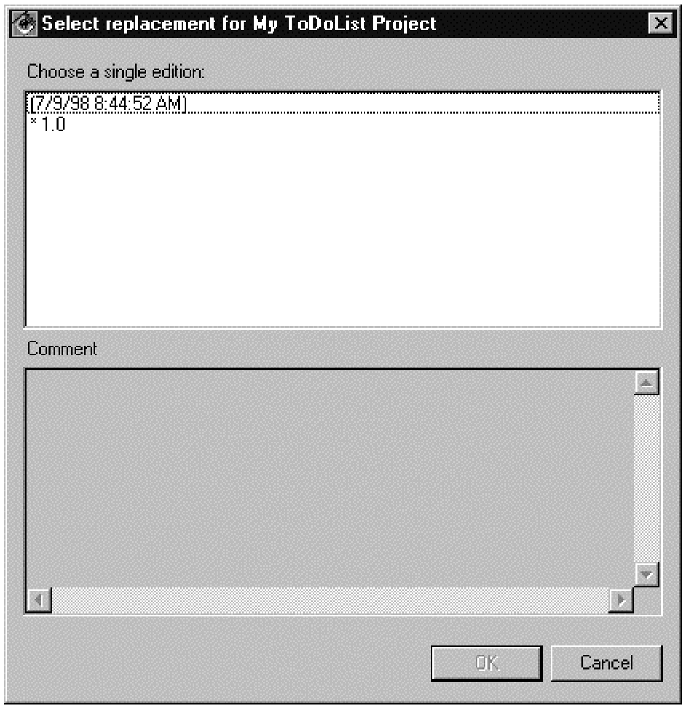
To see more information on the edition you're working on, use the Workbench window's tool bar to select **Show Edition Names**. Notice that each

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.

At any time, the workspace only contains one edition of a given program element: the edition that you are currently working on. To help manage your program elements, VisualAge for Java also includes a source code repository, which can contain many editions of the same program element. The workspace is the center of activity in the VisualAge for Java programming environment. The repository is not a development environment, but you can browse 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.

You can replace an edition that is in the workspace with another edition from the repository. 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.



(7/9/98 8:44:52 AM)
* 1.0

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`).

  The edition that is in the workspace may be a versioned edition, although any changes you make and save automatically create a new open edition.

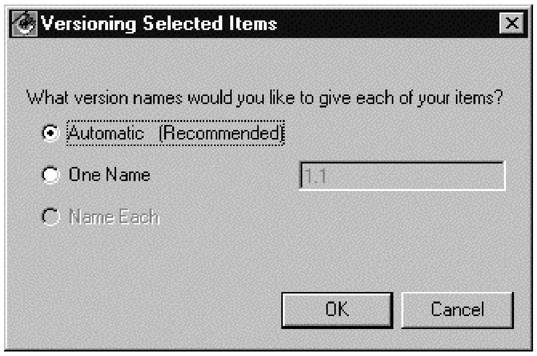
When you save a program element, not only is your code incrementally compiled behind the scenes, the open edition is updated in both the workspace and the repository.

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. From the**Selected** menu, click **Manage** and then **Version.** The Versioning Selected Items SmartGuide appears.

   What version names would you like to give each of your items?

   [✓ Automatic (Recommended)]

   One Name 1.1

   Name Each

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

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 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**.

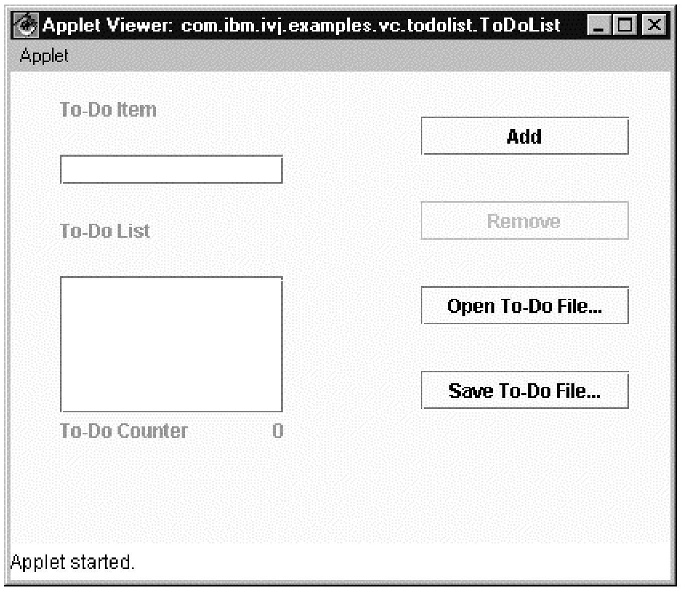
Notice that the edition name (in the hierarchy pane) changes from the versioned edition name to a timestamp. Because the workspace can only hold one edition of a program element at any given time, the new edition replaces the versioned edition. (Of course, a copy of the versioned edition can always be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

Adding a counter to the ToDoList program

Let’s add a counter to the ToDoList 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 running applet will look like this:



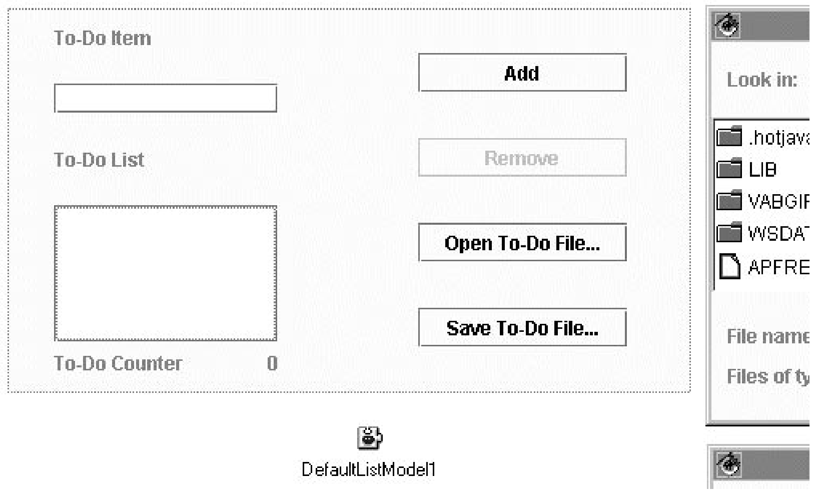
Adding the labels

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 a **JLabel** bean from the palette.
4. Click mouse button 1 beneath the list to add the label. You may wish to select the scrollpane, the text field and the labels and move them slightly up to make room for the new label.
5. Modify the text of the JLabel bean to **To-Do Counter**.
6. Add another JLabel bean to the right of the JLabel bean you just added.
7. Double-click on this new JLabel bean to open its Properties window. Select the value field to the right of the **horizontalAlignment** 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 new JLabel beans.
   - Select the counter name label, then the text field, and select the **Align Left** tool.

- Select the counter label, then the text field, and select the **Align Right** tool from the tool bar.
- 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.

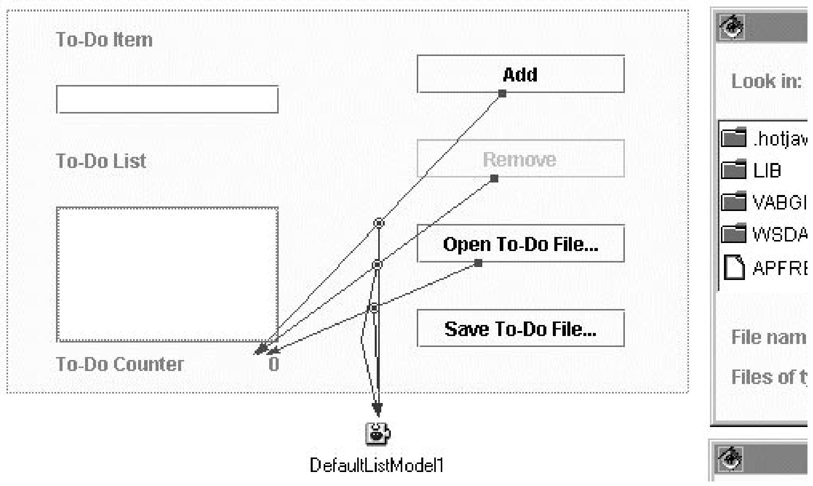
Connecting the labels

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**.
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 DefaultListModel bean and click mouse button 1.
6. From the pop-up menu, select **Connectable Features** to bring up the End connection to (DefaultListModel1) window.

7. From the **Method** list, select the **getSize()** method and then select **OK**. This provides the count of the list of items as input for setting the counter 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 count of items in the list whenever an action is taken that may modify the count. In this applet, any of the top three buttons have this potential.

Now the free-form surface should look like this:



From the **Bean** menu, select **Save Bean**. The changes you've made are reflected in this open edition, both in the workspace and in the repository. Select the **Run** 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, now you want to take out the counter code. Of course, you could just delete the labels and connections you've added, but you might 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 **Select replacement for ToDoList** 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.)

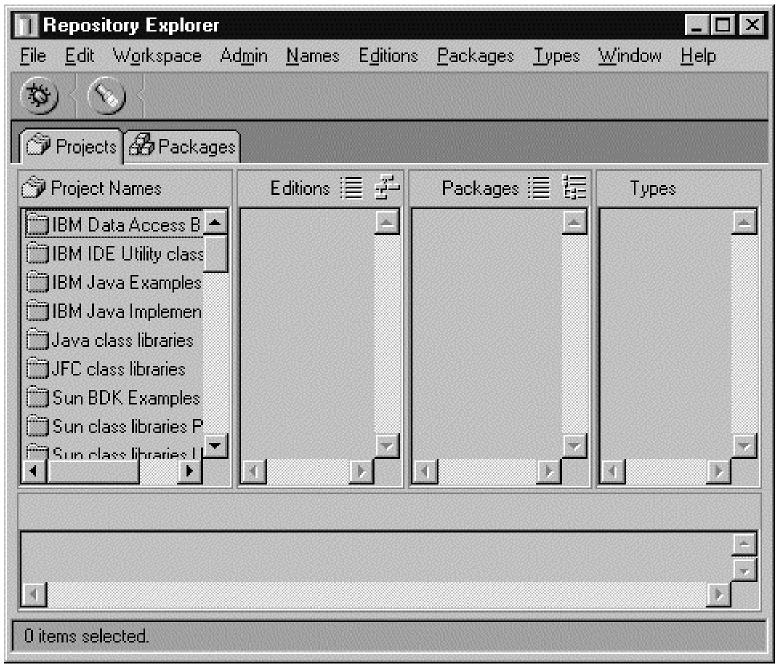
The edition information beside the class name now indicates the version number, not the timestamp of the open edition you had been working on.

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

In addition to its 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**.



IBM Data Access B
IBM IDE Utility class
IBM Java Examples
IBM Java Implemen
Java class libraries
JFC class libraries
Sun BDK Examples
Sun class libraries P
Sun class libraries U

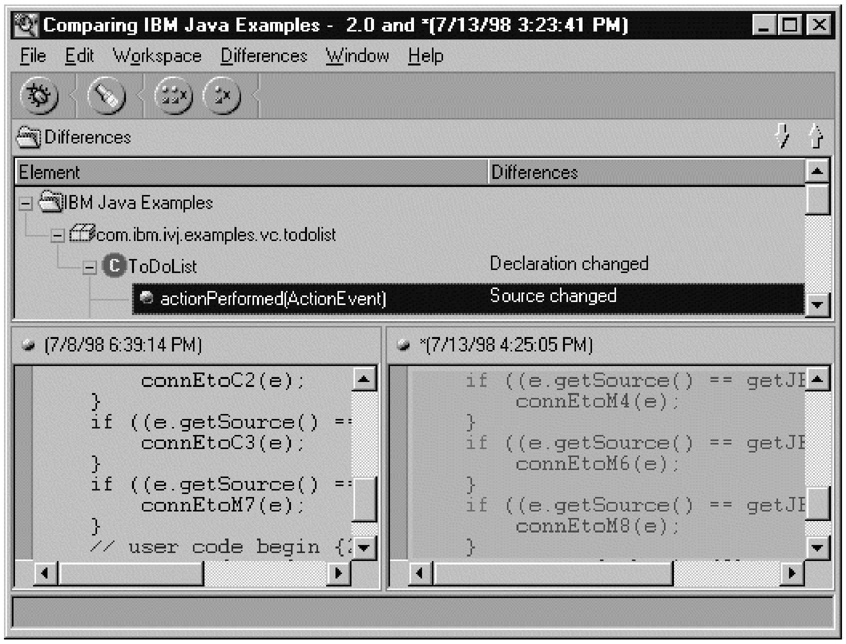
0 items selected.

The Repository Explorer provides a visual interface to 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 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 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 **Editions** menu, select **Compare**. The Comparing window appears:


Comparing IBM Java Examples - 2.0 and *(7/13/98 3:23:41 PM)
File Edit Workspace Differences Window Help
Differences

Element	Differences
IBM Java Examples	
com.ibm.ivj.examples.vc.todolist	
ToDoList	Declaration changed
actionPerformed(ActionEvent)	Source changed

(7/8/98 6:39:14 PM)

```
        connEtoC2(e);
    }
    if ((e.getSource() ==
        connEtoC3(e);
    }
    if ((e.getSource() ==
        connEtoM7(e);
    }
    // user code begin {
```

*(7/13/98 4:25:05 PM)

```
    if ((e.getSource() == getJ
        connEtoM4(e);
    }
    if ((e.getSource() == getJ
        connEtoM6(e);
    }
    if ((e.getSource() == getJ
        connEtoM8(e);
    }
```

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 List 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.todolist** from the **Packages** list. The class in this package appears in the **Types** list. To examine this class, select it 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. For example, to bring the completed version of the To-Do List program into the workspace:

1. In the Workbench, select the project into which you want to add the To-Do List program and select **Add** and then **Package** from the **Selected** menu. The Add Package SmartGuide appears.
2. Select **Add packages from the repository**.
3. Select **com.ibm.ivj.examples.vc.todolist** from the **Available package names** list. Select an edition from the **Editions list** and click **Finish**.

The package for the To-Do List 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 8. What Else Can You 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:

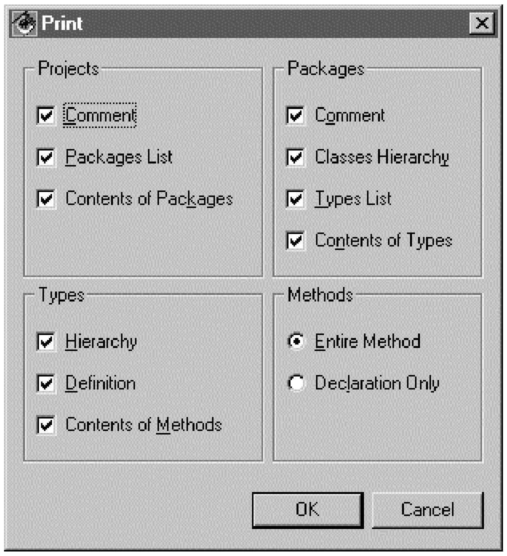
- Printing program elements
- Navigating
- Searching
- Browsing
- Writing code by hand
- Internationalization
- Using the Quick Start window
- Debugging
- Support for JavaBeans
- Customizing the workspace

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 **Document**, **Print** from its pop-up menu. The Print dialog.



Projects
- [✓] Comment
- [✓] Packages List
- [✓] Contents of Packages

Packages
- [✓] Comment
- [✓] Classes Hierarchy
- [✓] Types List
- [✓] Contents of Types

Types
- [✓] Hierarchy
- [✓] Definition
- [✓] Contents of Methods

Methods
- ● Entire Method
- ○ Declaration Only

2. 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 **Types** are available if you are printing a project, package, class, or interface.
   - Selections under **Methods** are always available.
3. By default, all the items under **Projects**, **Packages**, and **Types** are selected, and **Entire Method** is selected under **Methods**. Change these selections if you want and select **OK** to start printing.
4. If no default printer has been selected, a message appears asking you to select one.

Changing the default printer

You can change the default printer or change a printer's setup by selecting **Print Setup** from the **File** menu of any window.

Printing the Graph of a Class Hierarchy

You can print a graph view of a class hierarchy for a project or a package. The output goes to the default printer. To print a class hierarchy graph:

1. In the Workbench or another browser, select the project or package for which you want to print the graph.
2. From the element's pop-up menu, select **Open To**, **Classes**. This opens the Classes page of a browser on the project or package.

3. In the Class Hierarchy pane title bar, click the **Graph Layout** button .

4. From the Class Hierarchy pane's pop-up menu, select **Document**,**Print Graph**. The graph, showing the inheritance of each class, will be output to the default printer.

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.

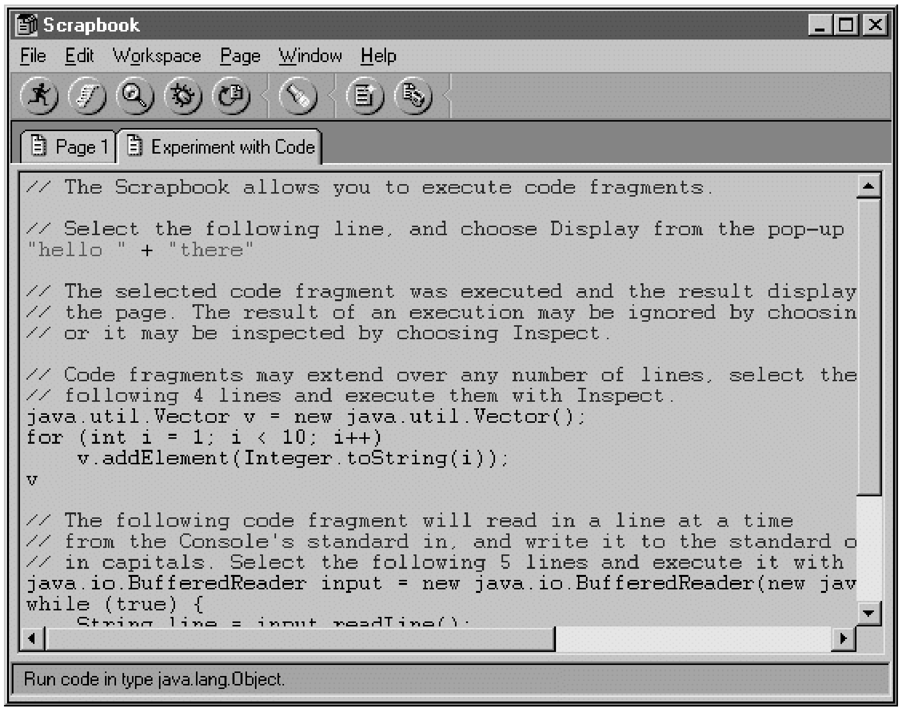
Recently-used windows are stored in a list in the **File** menu. If you want to open a window that you recently closed, select it from the list in the **File**menu.

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.

Windows you can open from the Window menu

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

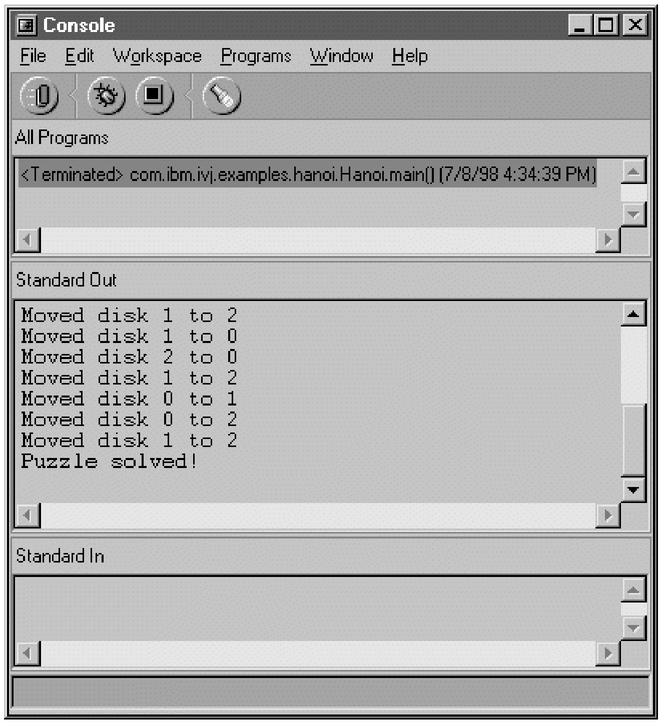
- Scrapbook - gives you a place to try out code. You can enter and run code fragments without making them a part of any package, project, or class.



```
// The Scrapbook allows you to execute code fragments.

// Select the following line, and choose Display from the pop-up
"hello " + "there"

// The selected code fragment was executed and the result display
// the page. The result of an execution may be ignored by choosin
// or it may be inspected by choosing Inspect.

// Code fragments may extend over any number of lines, select the
// following 4 lines and execute them with Inspect.
java.util.Vector v = new java.util.Vector();
for (int i = 1; i < 10; i++)
    v.addElement(Integer.toString(i));
v

// The following code fragment will read in a line at a time
// from the Console's standard in, and write it to the standard o
// in capitals. Select the following 5 lines and execute it with
java.io.BufferedReader input = new java.io.BufferedReader(new jav
while (true) {
    String line = input.readLine();
```

Run code in type java.lang.Object.

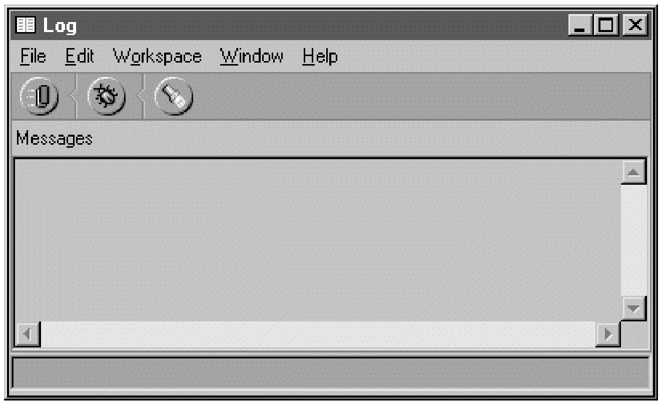
- Console - displays standard out. It also gives you an area for entering input to standard in. If more than one thread is waiting for input from standard in, you can select which thread gets the input.



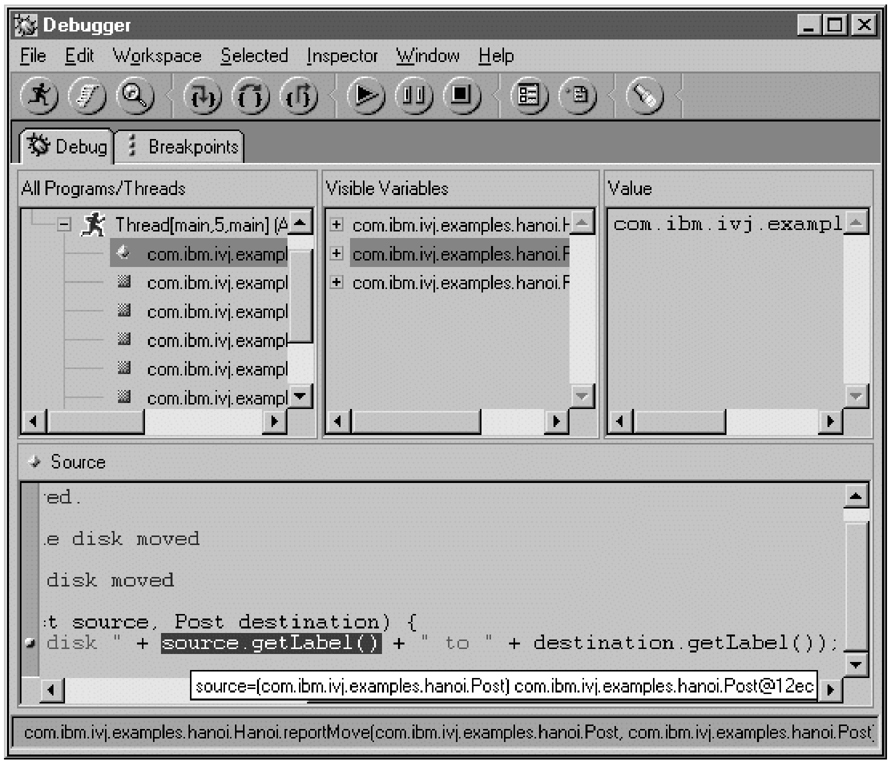
```
<Terminated> com.ibm.ivj.examples.hanoi.Hanoi.main() (7/8/98 4:34:39 PM)
```

```
Moved disk 1 to 2
Moved disk 1 to 0
Moved disk 2 to 0
Moved disk 1 to 2
Moved disk 0 to 1
Moved disk 0 to 2
Moved disk 1 to 2
Puzzle solved!
```

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



- Debugger - displays running threads and the contents of their runtime stacks. In the Debugger you can suspend and resume execution of threads, inspect and modify variable values, and set, remove, and configure breakpoints. The **Window** menu lets you open the Debugger browser to the Debug page or the Breakpoints page, or open the dialogs for setting breakpoints on external class methods and caught exceptions. See Debugging for more details.



```
Thread[main,5,main] (A
com.ibm.ivj.exampl
com.ibm.ivj.exampl
com.ibm.ivj.exampl
com.ibm.ivj.exampl
com.ibm.ivj.exampl
com.ibm.ivj.exampl
```

```
com.ibm.ivj.examples.hanoi.H
com.ibm.ivj.examples.hanoi.F
com.ibm.ivj.examples.hanoi.F
```

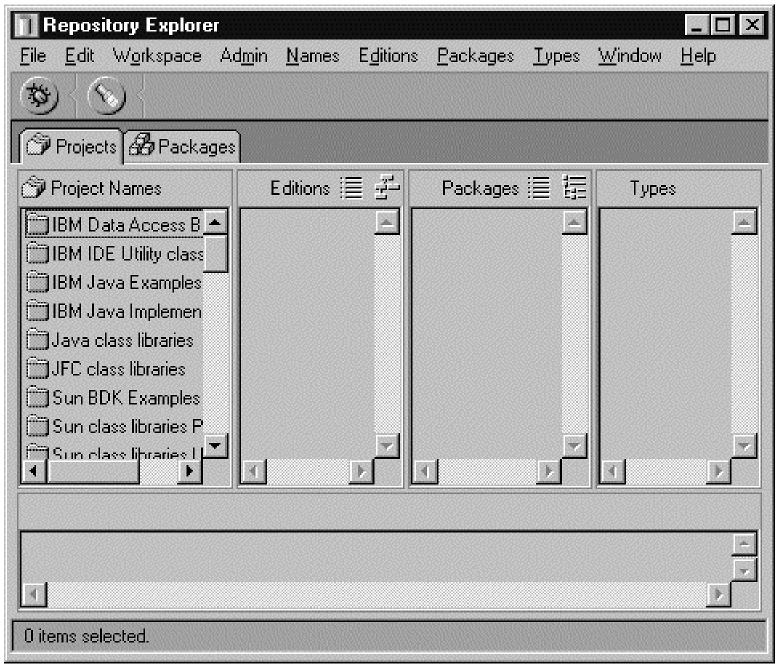
```
com.ibm.ivj.exampl
```

```
ed.

e disk moved

disk moved

t source, Post destination) {
disk " + source.getLabel() + " to " + destination.getLabel());
```

```
source=(com.ibm.ivj.examples.hanoi.Post) com.ibm.ivj.examples.hanoi.Post@12ec
```

```
com.ibm.ivj.examples.hanoi.Hanoi.reportMove(com.ibm.ivj.examples.hanoi.Post, com.ibm.ivj.examples.hanoi.Post)
```

- Repository Explorer - displays all of the editions of program elements in the repository. See Exploring the Repository for more details.



In addition to these windows, you can do the following actions from the Window menu.

- Clone - opens a duplicate of the current window. You can then browse the two windows independently. Changes made to program elements are reflected in both windows.
- Lock - locks open the current window. If you try to close it, a message box informs you that the window is locked. You must unlock the window before you can close it.
- Maximize - resizes the current window so that it covers the entire screen.
- Orientation - changes the general layout of the panes in the window. The images in this information, for example, show the horizontal orientation, which, in the IDE, you can optionally change to a vertical orientation.
- Show Edition Names - enables or disables edition name labels for program elements.
- Workbench - opens (or brings into focus) the Workbench browser.

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 the IDE.

Searching for a program element

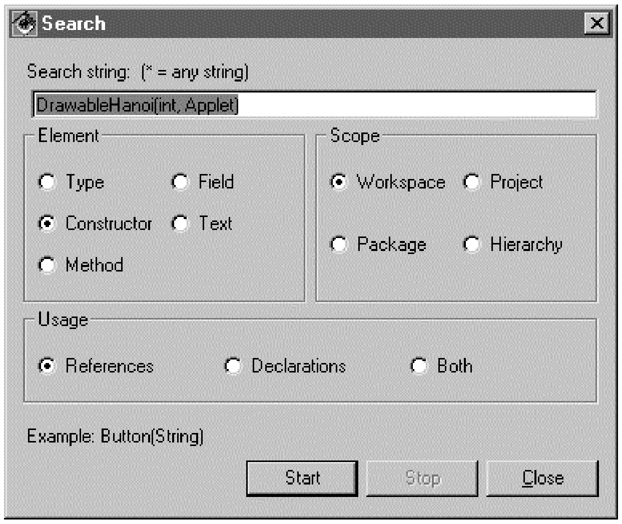
The IDE gives you several choices for searching for program elements. For example, in program element panes, if you press a letter key, VisualAge for Java selects the first displayed program element that begins with that letter. If you press the same letter again, the next program element that begins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

Searching with the Search dialog

You can use the Search dialog to perform powerful searches of the workspace. To open the Search dialog, use any one of the following methods:

- Select **Search** from the **Workspace** menu of any window.
- Select text in a Source pane, and then **Search** from the **Edit** menu.
- Select the **Search** button from any window's toolbar.

This will open the Search dialog. If you selected text or a program element before launching the search, the Search string field will contain what was selected.



Search string: (* = any string)

DrawableHanoi(int, Applet)

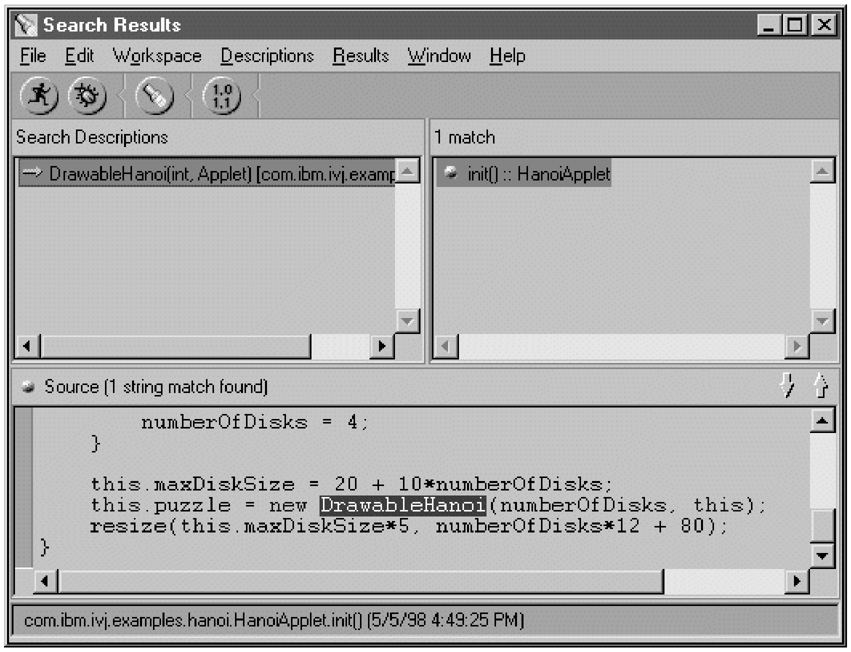
Element: Type Field [✓ Constructor] Text Method

Scope: [✓ Workspace] Project Package Hierarchy

Usage: [✓ References] Declarations Both

Example: Button(String)

Select the type of program element you want to search for, the scope of the search, and the usage of the element, by enabling the appropriate radio buttons. Click **Start**. When the search is complete, and if the IDE finds a match to your criteria, the Search Results window will open.


Search Results
File Edit Workspace Descriptions Results Window Help
Search Descriptions
DrawableHanoi(int, Applet) [com.ibm.ivj.examp
1 match
init() :: HanoiApplet
Source (1 string match found)

```
        numberOfDisks = 4;
    }

    this.maxDiskSize = 20 + 10*numberOfDisks;
    this.puzzle = new DrawableHanoi(numberOfDisks, this);
    resize(this.maxDiskSize*5, numberOfDisks*12 + 80);
}
```

com.ibm.ivj.examples.hanoi.HanoiApplet.init() (5/5/98 4:49:25 PM)

In the Search Results window, you can browse and modify the contained program elements, and re-run searches.

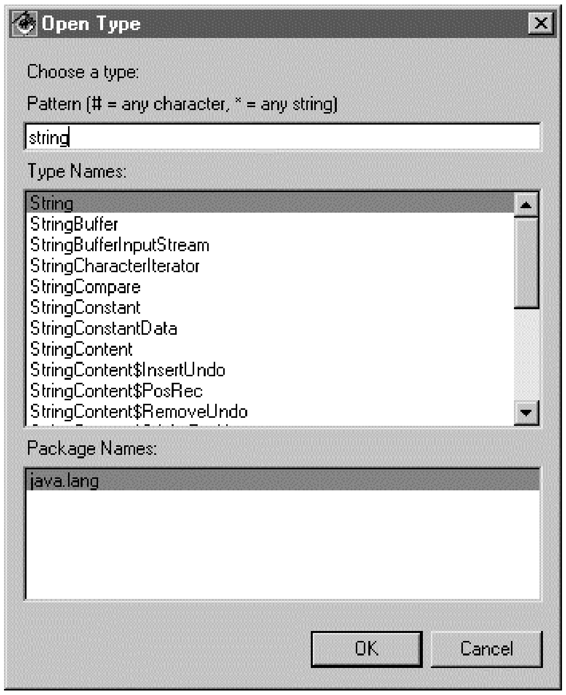
Searching for references and declarations

The pop-up menu for types and methods contains special searches that are often needed. For classes and interfaces, The **References To** pop-up menu option has sub-options that search the workspace for references to the selected type or one of its fields. Results of the search are displayed in the Search Results window.

For methods, the **References To** pop-up menu option has sub-options that search the workspace for references to the selected method, methods it calls, fields it accesses, or types it references. The **Declarations Of** sub-options search for declarations of these same program elements. Results of the search are displayed in the Search Results window.

Searching from the Workspace menu

You can also search for a program element by selecting one of the **Open** selections from the **Workspace** menu. Running this search will result in opening a browser on the searched-for element. For example, if you select **Open Type Browser** from the **Workspace** menu, the Open Type dialog is displayed:



As you enter *string* in the **Pattern** field, the **Type Names** list updates to show only the classes and interfaces that match what you have typed in so far. Select String from the **Type Names** list and select **OK** to open a browser on the String class. For some type names, which may exist in more than one package, you also need to select a package from the **Package Names** list.

Searching for a program element within a browser page

If you want to find a program element that you know is contained in the current browser page, use the **Go To** menu option for the element type. For example, if you are in the Projects page of the workbench, and you want to find the java.lang.String class, Select **Go To**, **Type** from the **Selected** menu. In the Go To Type dialog, enter *string* in the **Pattern** field, select the String class from the type list, and java.lang from the package list. When you click **OK**, the IDE will go to and select the java.lang.String class in the All Projects pane in the Workbench.

Browsing

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

In the IDE, 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:

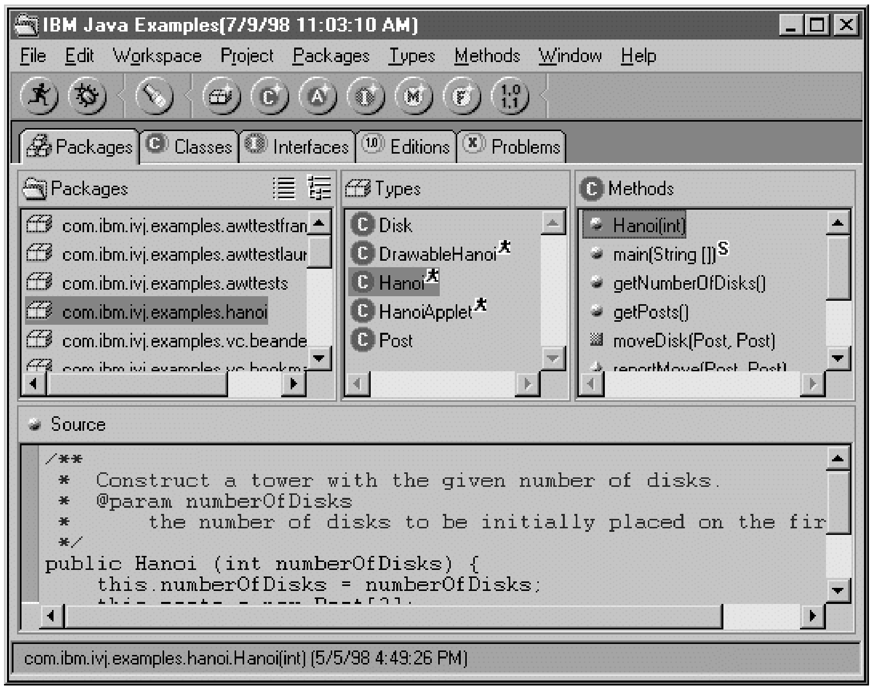
- Select the program element and select **Open** from the **Selected** menu or from the pop-up menu for the program element.
- Select the appropriate browser in the **Workspace** menu (**Open Type Browser**, **Open Package Browser**, or **Open Project Browser**) for the program element. A secondary window 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** page displays all the editions of this project.
- The **Problems** page lists all program elements in the project that contain errors.


IBM Java Examples(7/9/98 11:03:10 AM)
File Edit Workspace Project Packages Types Methods Window Help
Packages Classes Interfaces Editions Problems
Packages
Types
Methods
Source

com.ibm.ivj.examples.awttestfran
com.ibm.ivj.examples.awttestlaur
com.ibm.ivj.examples.awttests
com.ibm.ivj.examples.hanoi
com.ibm.ivj.examples.vc.beande

Disk
DrawableHanoi
Hanoi
HanoiApplet
Post

Hanoi(int)
main(String [])
getNumberOfDisks()
getPosts()
moveDisk(Post, Post)

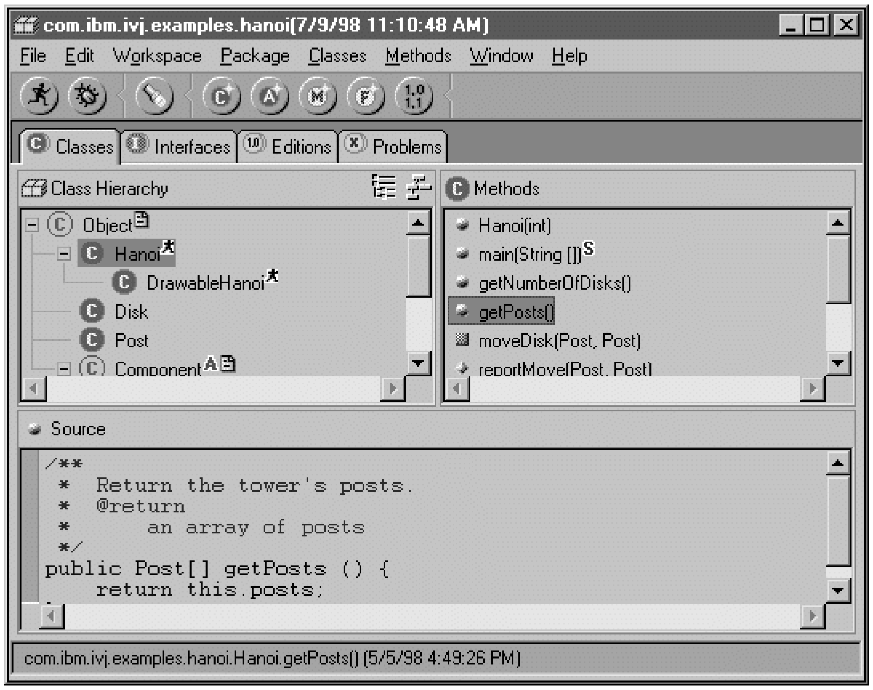
```
/**
 *  Construct a tower with the given number of disks.
 *  @param numberOfDisks
 *      the number of disks to be initially placed on the fir
 */
public Hanoi (int numberOfDisks) {
    this.numberOfDisks = numberOfDisks;
```

com.ibm.ivj.examples.hanoi.Hanoi(int) (5/5/98 4:49:26 PM)

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** page displays all the editions of this package
- The **Problems** page lists all program elements in the package that contain errors.



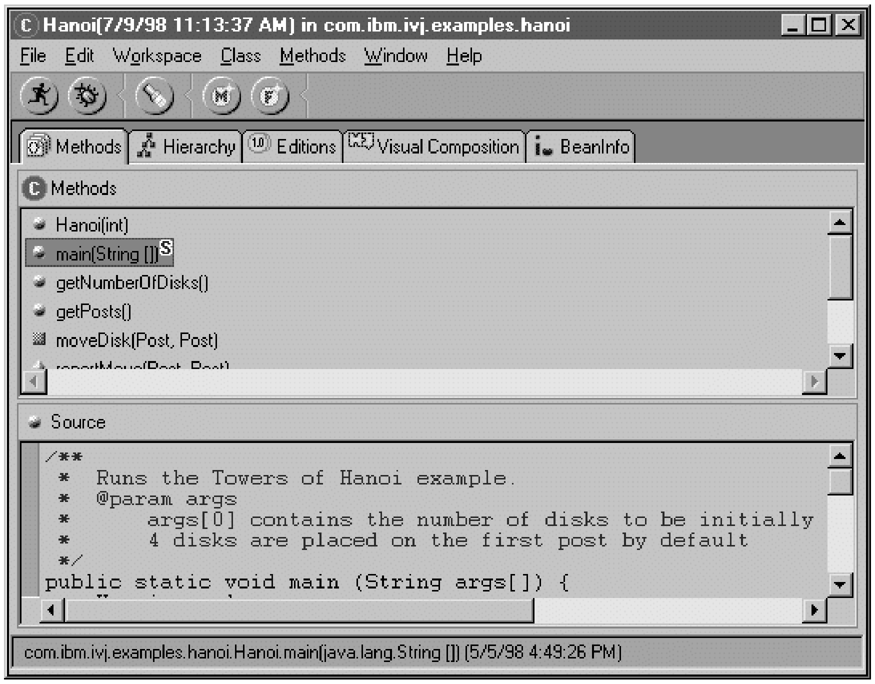
```
/**
 *  Return the tower's posts.
 *  @return
 *      an array of posts
 */
public Post[] getPosts () {
    return this.posts;
```

com.ibm.ivj.examples.hanoi.Hanoi.getPosts() (5/5/98 4:49:26 PM)

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** page displays the editions of this class.
- The **Visual Composition** page displays the Visual Composition Editor.
- The **BeanInfo** page displays the JavaBean information for this class.


Hanoi(7/9/98 11:13:37 AM) in com.ibm.ivj.examples.hanoi
File Edit Workspace Class Methods Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Methods
Hanoi(int)
main(String [])
getNumberOfDisks()
getPosts()
moveDisk(Post, Post)
Source

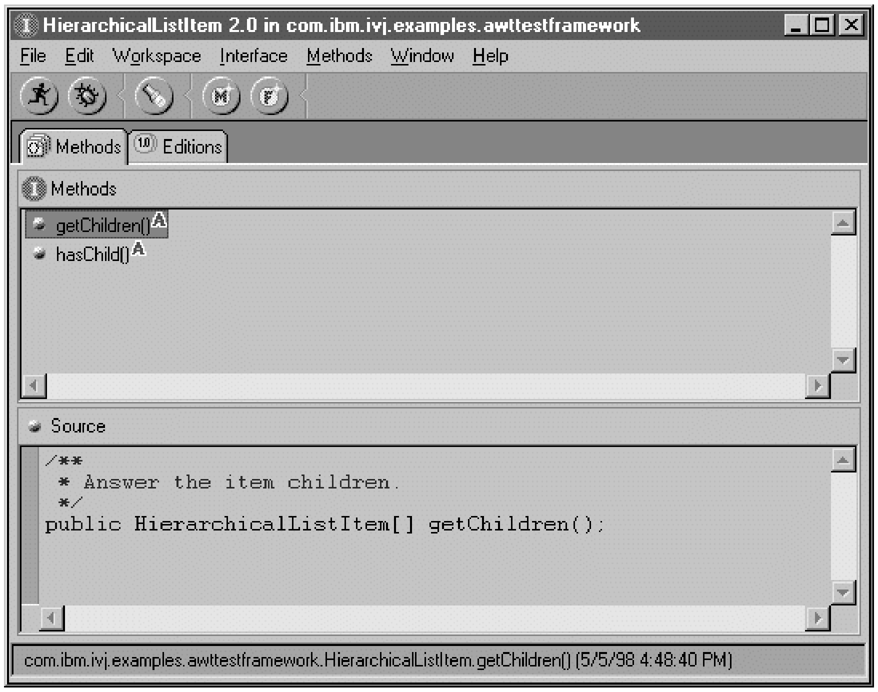
```
/**
 *  Runs the Towers of Hanoi example.
 *  @param args
 *      args[0] contains the number of disks to be initially
 *      4 disks are placed on the first post by default
 */
public static void main (String args[]) {
```

com.ibm.ivj.examples.hanoi.Hanoi.main(java.lang.String []) (5/5/98 4:49:26 PM)

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** page displays the editions of this interface


HierarchicalListItem 2.0 in com.ibm.ivj.examples.awttestframework
File Edit Workspace Interface Methods Window Help
Methods
Editions
Methods
getChildren()
hasChild()
Source

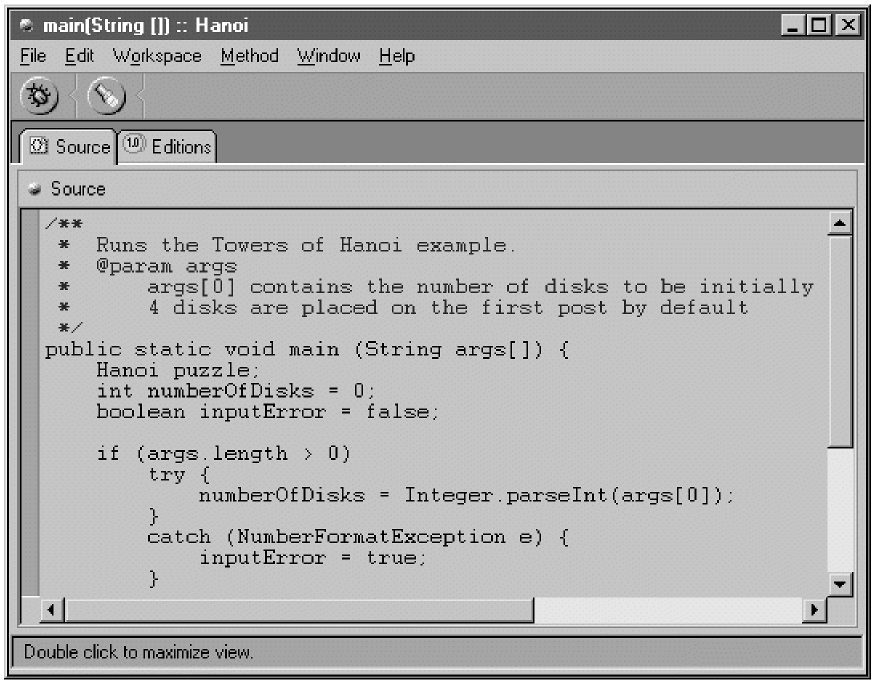
```
/**
 * Answer the item children.
 */
public HierarchicalListItem[] getChildren();
```

com.ibm.ivj.examples.awttestframework.HierarchicalListItem.getChildren() (5/5/98 4:48:40 PM)

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** page displays the editions of this method


main(String []) :: Hanoi
File Edit Workspace Method Window Help
Source
Editions
Source

```
/**
 *  Runs the Towers of Hanoi example.
 *  @param args
 *      args[0] contains the number of disks to be initially
 *      4 disks are placed on the first post by default
 */
public static void main (String args[]) {
    Hanoi puzzle;
    int numberOfDisks = 0;
    boolean inputError = false;

    if (args.length > 0)
        try {
            numberOfDisks = Integer.parseInt(args[0]);
        }
        catch (NumberFormatException e) {
            inputError = true;
        }
```

Double click to maximize view.

Writing code by hand

For the most part, as you have progressed through this *Getting Started* document, you have been using the Visual Composition Editor and the SmartGuides to generate Java code for you, or you have been copying in sections of code that we have provided for you. When you create your own applications, you will likely need to write sections of code by hand, in the Source panes of the IDE browsers. VisualAge for Java provides several tools to help you write correct, neat code by hand. This section describes some of these tools.

Code Assist

Source panes and some other dialogs and browsers (for example, the Configure Breakpoints dialog) contain *code assist*, a tool to help you find the classes, methods, and fields you are looking for without having to refer to class library reference information. Code assist is accessed by typing Ctrl+Spacebar.

When you type Ctrl+Spacebar, classes methods, and types that could be inserted in the code at the cursor are shown in a pop-up list, from which you can select one. Code assist performs a visibility check and classes, methods, and fields that are not visible are not displayed. If the code assist mechanism cannot find a member that fits the current location of the cursor, the information line at the bottom of the pane will indicate that no code assist is available for the current context.

Code Assist for Types

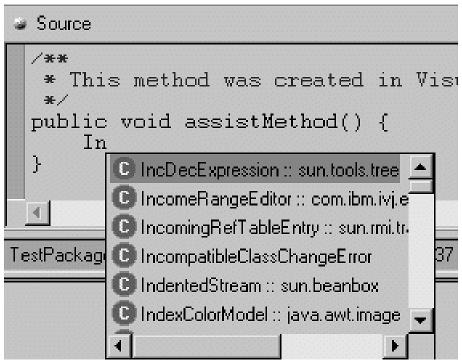
To insert the name of a class or interface in your code, enter the first one or more letters of the type name, and then type Ctrl+Spacebar. A pop-up list appears, containing types that start with what you have entered. Enter more letters to narrow down the list. Select an item to insert it into your code at the cursor. If the type needs to be qualified, the qualification is also automatically inserted.

Example:

Create a test project and package. In the test package, create a class called AssistTest. In the AssistTest class, create a method called assistMethod. Suppose you want to declare a local Integer variable, `i`. In the body of the assistMethod source, type the following letters:

```
In
```

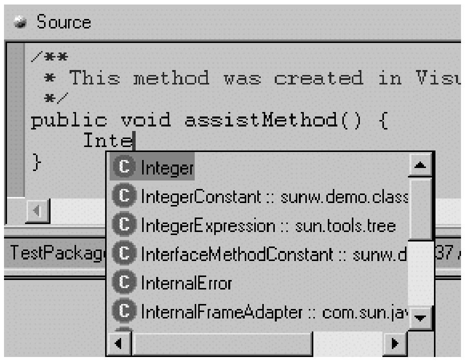
Type Ctrl+Spacebar. The pop-up list of options will appear.



```
/**
 * This method was created in Visu
 */
public void assistMethod() {
    In
}
```

Source
IncDecExpression :: sun.tools.tree
IncomeRangeEditor :: com.ibm.ivj.e
IncomingRefTableEntry :: sun.rmi.tr
IncompatibleClassChangeError
IndentedStream :: sun.beanbox
IndexColorModel :: java.awt.image
TestPackag
37

The list of available types is long. To find 'Integer,' enter the letters 'te'. Now, 'Integer' will be near the top of the list.



```
/**
 * This method was created in Visu
 */
public void assistMethod() {
    Inte
}
```

Integer
IntegerConstant :: sunw.demo.class
IntegerExpression :: sun.tools.tree
InterfaceMethodConstant :: sunw.d
InternalError
InternalFrameAdapter :: com.sun.jav
TestPackag

Select it using the arrow keys and Enter.

Now, finish the declaration, so that the method looks like this:

```
public void assistMethod() {
    Integer i;
}
```

Save the method by typing Ctrl+S. You will use this test method in the next example.

Code Assist for Methods and Fields

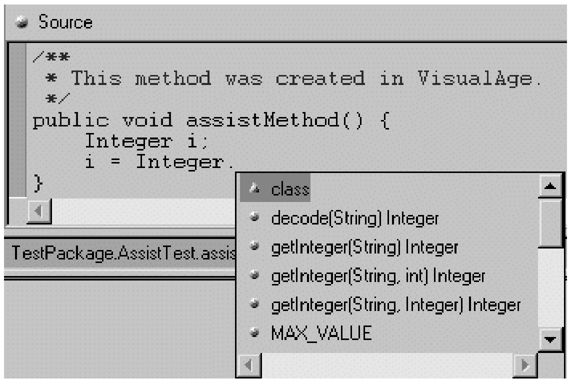
Code assist will also list the methods and fields available for an object or class. Enter *objectName*., and optionally one or more letters from the start of the method or field name, and then type Ctrl+Spacebar. The list of methods and fields for the object will pop-up. Select one to insert it in the code.

Example:

In the assistMethod method you created in the previous example, below the line that declares `i`, enter the following code (the period is important):

```
i = Integer.
```

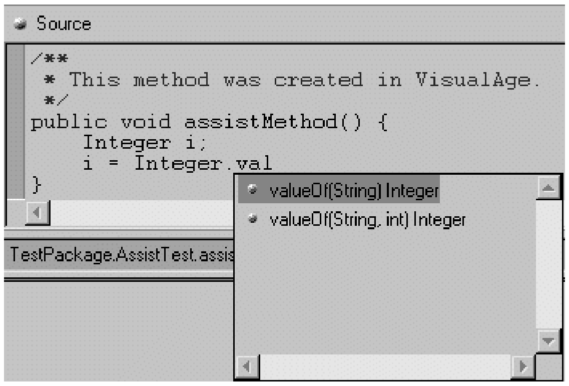
Type Ctrl+Spacebar. A list of methods and fields in Integer will pop up.



```
/**
 * This method was created in VisualAge.
 */
public void assistMethod() {
    Integer i;
    i = Integer.
}
```

Source
TestPackage.AssistTest.assis
class
decode(String) Integer
getInteger(String) Integer
getInteger(String, int) Integer
getInteger(String, Integer) Integer
MAX_VALUE

Enter the letters 'val', until you find 'valueOf(String) Integer'. The parameter types (in this case 'String') and return type ('Integer') are shown.



```
/**
 * This method was created in VisualAge.
 */
public void assistMethod() {
    Integer i;
    i = Integer.val
}
```

Select 'valueOf(String) Integer', and it will be inserted into your code. Enter a string such as `'35'` between the parentheses and end the line with a semi-colon. The method source will now look like this:

```
public void assistMethod() {
    Integer i;
    i = Integer.valueOf('35');
}
```

If you request code assist for a method or field from a class that requires qualification, the class must be qualified before you type Ctrl+Spacebar. Otherwise, no code assist will be available. Generally, the code that appears before the cursor must be compilable before you request code assist.

Example:

Suppose java.util.* is not in your class' import statement. This means that the class ResourceBundle must be qualified when you use it in your class. If you type the following code, and then type Ctrl+Spacebar to get the list of methods available, no list will be available:

```
public String newMethod ( ) {
ResourceBundle a = ResourceBundle.
                      // place cursor after period
                      // and type Ctrl+Spacebar
```

However, if you type the following code, where the class qualification is provided, code assist is available:

```
public String newMethod ( ) {
ResourceBundle a = java.util.ResourceBundle.
                      // place cursor after period
                      // and type Ctrl+Spacebar
```

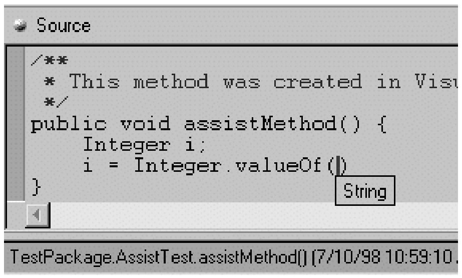
An easier way to produce a qualified name in this case (assuming you do not want to add this class or package to the import list) is to place the cursor *before* the period and type Ctrl+Spacebar. Select the class name from the list and it will be fully qualified for you automatically. Then type the period and Ctrl+Spacebar. The list of methods in ResourceBundle will now pop-up.

Code Assist for Method Parameters

Code assist includes pop-up help for method parameters. For example when you select 'valueOf(String) Integer' from the pop-up list in an example, above, the following text is inserted at the cursor:

```
valueOf()
```

The cursor is automatically placed between the parentheses, and the pop-up label 'String' appears to let you know what type of parameter to add.



```
/**
 * This method was created in Visu
 */
public void assistMethod() {
    Integer i;
    i = Integer.valueOf()
}
```

TestPackage.AssistTest.assistMethod() (7/10/98 10:59:10.

Important to Note:

- Code assist is case sensitive. For example, if you are looking for a program element that starts with upper-case 'C', then ensure you type in an upper-case letter 'C' before you type Ctrl+Spacebar. Likewise, as you narrow down the pop-up list, type in the proper case.
- Code assist is not available in a class definition.

- When accessing code assist for names that start with Java keywords such as 'for', 'while', or 'if', type Ctrl+Spacebar at least one letter *before or after* the end of the keyword. Otherwise, the information bar will indicate that no code assist is available in the current context.
- If typing Ctrl+Spacebar does not launch code assist on your system, try using Ctrl+L.

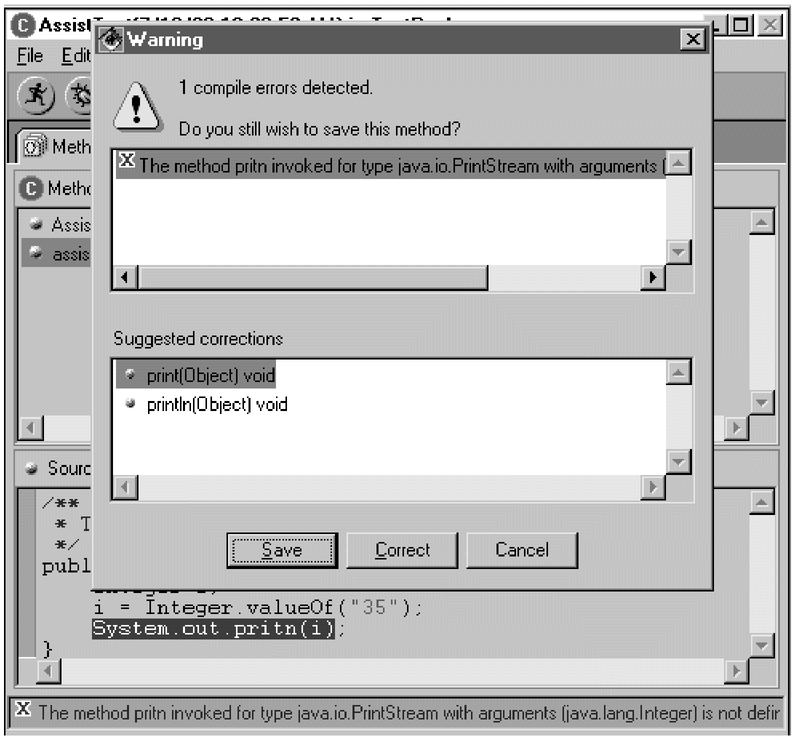
Code Clues

If you try to save code that contains an error, the IDE warns you that the code has an error. If it can determine the type of error, it will present a list of possible solutions. You can select one and correct the error, or you can save the code with the error (it will be added to the list of problems on the Problem page of the IDE browsers that contain the program element).

For example, add the following line (including the mistake) to the assistMethod method from above:

```
System.out.pritn(i);
```

When you save the method, the following dialog will appear, suggesting alternative code that will fix the problem:



1 compile errors detected.

Do you still wish to save this method?

The method pritn invoked for type java.io.PrintStream with arguments (

print(Object) void
println(Object) void

```
i = Integer.valueOf("35");
System.out.pritn(i);
}
```

The method pritn invoked for type java.io.PrintStream with arguments (java.lang.Integer) is not defir

Select the suggested correction 'print(Object) void' and click **Correct**. The method will be saved with the replacement code. If you click **Save**, the method will be saved with the error. If you click **Cancel**, the method will not be saved and the error will remain in the code.

Format Code

To promote neat, easy-to-read coding, the IDE provides an automatic code formatter which automatically controls how your code appears when you write it in a Source pane. To set code formatter options, including indentation and new-line controls:

1. Open the Options dialog by selecting **Options** from the **Window** menu.
2. In the left-hand list in the Options dialog, expand the **Coding** item.
3. Select the **Formatter** item. On the Formatter page you can enable options that tell the Source panes to start a new line for each statement in a compound statement, or to use and opening brace.
4. Select the **Indentation** item. On the Indentation page, you can select an indentation style.

These specifications are applied automatically to all new code. If you have imported code from the file system, or if you change the formatting options, you can apply the options to code in a particular source pane by selecting **Format Code** from the pane’s pop-up menu.

Internationalization Support

VisualAge for Java supports two means of text separation by locale: list bundles and property files. A list bundle is a persistent form of *java.util.ListResourceBundle*. A property file is a persistent form of *java.util.PropertyResourceBundle*.

Both types of resource bundle contain key-value pairs. For list resource bundles, these pairs are stored within a bundle class in the repository: *ListResourceBundle.getContents( )* returns an array of key-value pairs. The key-value pairs in a property resource bundle are stored on the file system.

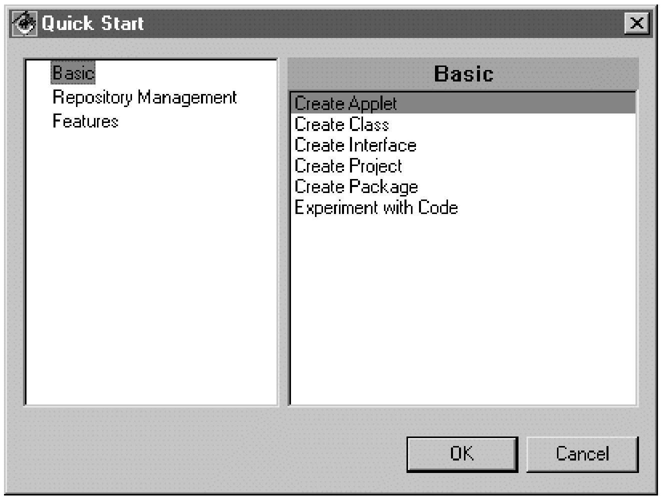
Each resource bundle contains values for one (or a default) locale. The name of the bundle can be keyed by locale so that the virtual machine loads the appropriate resources for the current locale setting.

VisualAge for Java supports the creation, editing, and use of resource bundles for all text found in a class. You can separate String property values as you set them from the Visual Composition Editor, or you can separate all text at once from the Workbench.

You can use your own resource bundles, or you can create them using VisualAge for Java. You can edit existing resource bundles by hand or from within VisualAge for Java. Multiple resource sources can be referenced within a single bean. VisualAge for Java generates the appropriate code the next time you save the bean.

Using the Quick Start window

Several of the most frequent tasks you will perform in the IDE are collected in an easy-to-access window called Quick Start. To open the Quick Start window, select Quick Start from any IDE window’s **File** menu, or press F2.



The list on the left-hand side shows the categories of Quick Start Tasks. Select one to see the available tasks, displayed in the right-hand side. To start a task, select it and click **OK**.

Basic Tasks

The basic tasks for creating program elements launch the appropriate SmartGuide to help you create applets, classes, interfaces, projects, and packages.

The Experiment with Code task opens a page in the Scrapbook. The page provides introductory instructions for learning what the Scrapbook can do. It guides you through learning the basic steps to evaluating code and variable values in the Scrapbook.

Team Development Tasks [ENTERPRISE] The team development tasks provide you with easy access to projects in the repository. Also, a team administrator can administer users. The **Management Query** option lets you search for program elements based on status, owner, or developer.

Repository Management Tasks

The **Compact Repository** task lets you duplicate the current repository, leaving out purged and opened editions.

[ENTERPRISE] You can use the **Change Repository** option to connect your workspace to another repository (shared or local), or to recover if the server fails.

Features Tasks

With the **Add Feature** and **Remove Feature** options, you can add to your workspace projects that are provided with VisualAge for Java, but are installed as part of the repository. These projects include IDE samples and Data Access Beans. A secondary window will appear to let you select which of these projects ('features') to add or remove.

Debugging

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

Opening the debugger

You can open the debugger manually by selecting **Debug**, **Debugger** from the **Window** menu. If a program is running, you can suspend its thread and view its stack and variable values. Alternatively, the debugger will automatically open, with the current thread suspended, for any of several reasons:

- A breakpoint in the code is encountered.
- A conditional breakpoint that evaluates to true is encountered.
- An exception is thrown and not caught.
- An exception selected in the Caught Exceptions dialog is thrown.
- A breakpoint in an external class is encountered.

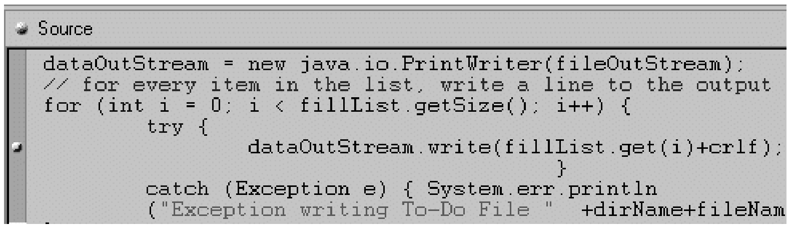
Setting breakpoints

When a program is running in the IDE and encounters a breakpoint, the running thread is suspended and the Debugger browser is opened so that you can work with the method stack and inspect variable values. In the IDE, 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 ToDoList class from the To-Do List program.

To set this breakpoint:

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

   ```
   dataOutStream.write(fillList.get(i)+crlf);
   ```
4. A breakpoint indicator appears in the margin of the Source pane beside this line:



```
dataOutStream = new java.io.PrintWriter(fileOutStream);
// for every item in the list, write a line to the output
for (int i = 0; i < fillList.getSize(); i++) {
        try {
                dataOutStream.write(fillList.get(i)+crlf);
                                                        }
        catch (Exception e) { System.err.println
        ("Exception writing To-Do File "  +dirName+fileNam
```

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 **Breakpoint** from the pop-up menu.

Removing breakpoints

To remove a breakpoint in a source pane, double-click 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 **Breakpoint** from the pop-up menu.

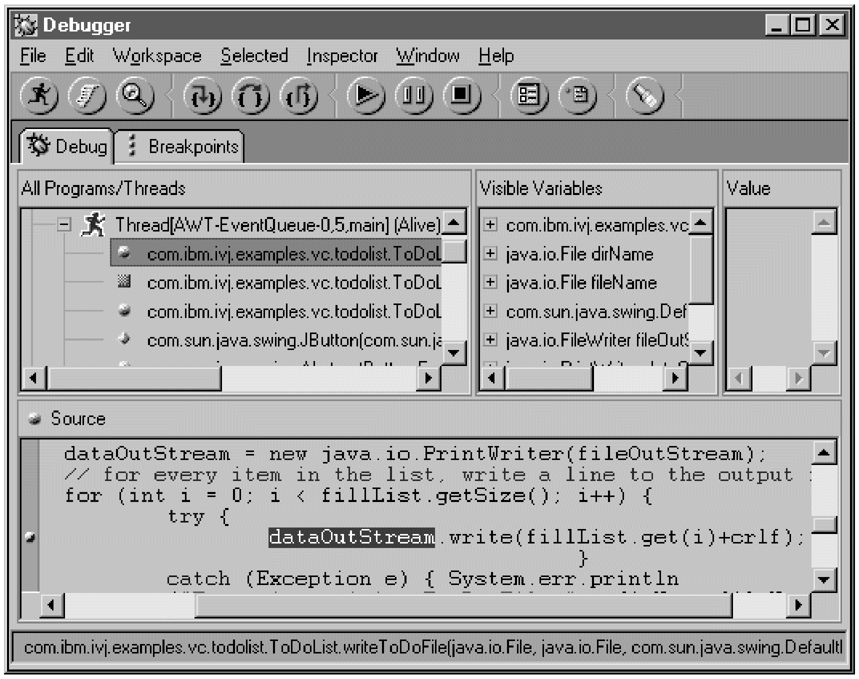
Try removing the breakpoint you just set. Now reset it. You will be using this breakpoint in the next section to examine the features of the Debugger browser.

Using the Debugger browser

The Debugger browser 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 List program to see what happens:

1. In the Workbench, select the ToDoList class. Select the **Run** toolbar button .
2. When the To-Do File program appears, add at least three 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 browser appears. It should look like this:

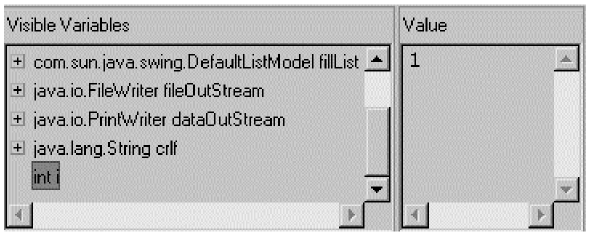

Debugger
File Edit Workspace Selected Inspector Window Help
Debug
Breakpoints
All Programs/Threads
Thread[AWT-EventQueue-0,5,main] (Alive)
com.ibm.ivj.examples.vc.todolist.ToDoL
com.ibm.ivj.examples.vc.todolist.ToDoL
com.ibm.ivj.examples.vc.todolist.ToDoL
com.sun.java.swing.JButton(com.sun.ja
Visible Variables
com.ibm.ivj.examples.vc
java.io.File dirName
java.io.File fileName
com.sun.java.swing.Def
java.io.FileWriter fileOut
Value
Source

```
dataOutStream = new java.io.PrintWriter(fileOutStream);
// for every item in the list, write a line to the output :
for (int i = 0; i < fillList.getSize(); i++) {
        try {
                dataOutStream.write(fillList.get(i)+crlf);
                                        }
        catch (Exception e) { System.err.println
```

com.ibm.ivj.examples.vc.todolist.ToDoList.writeToDoFile(java.io.File, java.io.File, com.sun.java.swing.Defaultl

   The thread you are debugging is selected in the All Programs/Threads list. The list of methods below the thread is the current stack. When you select a method in the stack the Visible Variables pane shows its visible variables. The Source pane shows the source where the breakpoint is set.

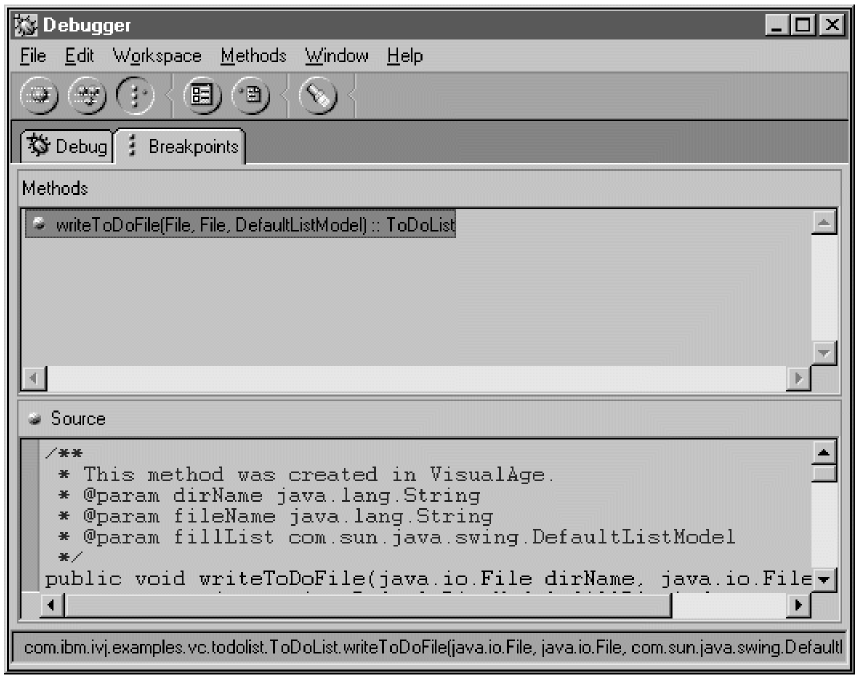
3. Select the **Resume** toolbar button to continue execution of the program. Because this breakpoint is inside a loop that writes each item to the file, the thread is suspended again and the Debugger window displays it.
4. Examine some of the variables in the Visible Variables list. For example, to see the value of the loop counter variable *i*, select **int i** from the Visible Variables pane (it’s at the bottom of the list). Its value appears in the Value pane:



```
com.sun.java.swing.DefaultListModel fillList    1
java.io.FileWriter fileOutStream
java.io.PrintWriter dataOutStream
java.lang.String crlf
int i
```

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** tab in the Debugger browser. The Breakpoints page appears:



writeToDoFile(File, File, DefaultListModel) :: ToDoList

```
/**
 * This method was created in VisualAge.
 * @param dirName java.lang.String
 * @param fileName java.lang.String
 * @param fillList com.sun.java.swing.DefaultListModel
 */
public void writeToDoFile(java.io.File dirName, java.io.File
```

com.ibm.ivj.examples.vc.todolist.ToDoList.writeToDoFile(java.io.File, java.io.File, com.sun.java.swing.Defaultl

   - The Breakpoints page lists 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, click the **Global Enable Breakpoints** tool bar button so that it is in the up (disabled) position. The breakpoint indicator changes colors to show that it is disabled. Note that it is not removed, but it will be ignored when the program resumes.
   - Select the **Debug** tab to return to the Debug page.

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.getSize()-1; i >= 0; i–) {
   ```

   Make this change in the Source pane of the Debugger page, 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 page and enable your breakpoints again by clicking the **Global Enable Breakpoints** button into the down position.

Other things you can do with the integrated debugger

The debugger has many other features that you will find helpful for debugging your programs. To learn more about the following tasks, as well as others, see the online help for the integrated debugger.

Set conditional breakpoints

Sometimes you want a breakpoint to suspend the thread only under certain conditions. A breakpoint can be configured so that an expression is evaluated before the debugger decides to suspend execution. If the expression includes a boolean that evaluates to true, the breakpoint suspends execution as usual. If it evaluates to false, the breakpoint is ignored.

To configure a breakpoint, click mouse button 2 on its symbol in the margin of a source pane. Select **Modify** from the pop-up menu. Enter the expression in the field. See the online help for the integrated debugger for more details on configuring breakpoints.

Set external and caught exception breakpoints

Besides setting breakpoints in code in the workspace, you can also set breakpoints on methods in external classes (classes that reside outside the workspace, in the file system, and that are loaded at run-time). You can also specify exception types that will break execution if they are thrown, even if your code catches and handles them. See the online help for the integrated debugger for more details on external breakpoints and breakpoints on caught exceptions.

Step through code

When a running thread has been suspended, you can step through code line by line or method by method, in a variety of ways. This is a controlled way of checking variable values at each point in your program.

Using inspectors to view and modify variables

You can open an inspector window to look closely at a particular variable in a suspended thread. The inspector lets you view and modify variable values and evaluate expressions.

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 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 might 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. 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, written, or both. 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 (triggers the vetoableChange event).
   - An **indexed** property is an array, so it exposes additional methods to address individual elements.
   - A **hidden** property is not visible to humans. It is for use by bean-aware tools only.
   - An **expert** property should only be manipulated by expert users.
   - A **normal** property is one that is neither hidden nor expert.

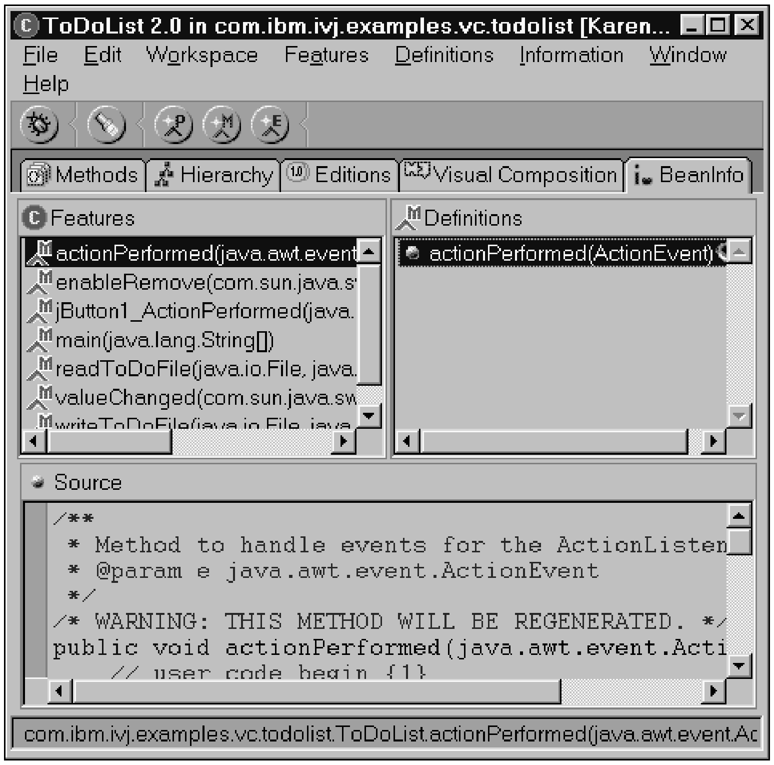
BeanInfo Classes

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.


ToDoList 2.0 in com.ibm.ivj.examples.vc.todolist [Karen...
File Edit Workspace Features Definitions Information Window Help
Methods
Hierarchy
Editions
Visual Composition
BeanInfo
Features
Definitions
Source

actionPerformed(java.awt.event
enableRemove(com.sun.java.s
jButton1_ActionPerformed(java.
main(java.lang.String[])
readToDoFile(java.io.File, java.
valueChanged(com.sun.java.sw
writeToDoFile(java.io.File, java

actionPerformed(ActionEvent)

```
/**
 * Method to handle events for the ActionListen
 * @param e java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
public void actionPerformed(java.awt.event.Acti
    // user code begin {1}
```

com.ibm.ivj.examples.vc.todolist.ToDoList.actionPerformed(java.awt.event.Ac

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? The IBM Java Examples package *com.ibm.ivj.examples.vc.customerinfo* is an example that makes use of property features. For instance, the Address class has properties for *street, city, state*, and *zipCode*. For instructions on how to build this sample, go to the Samples portion of the online product documentation under the Visual Composition samples and select CustomerInfo.

Customizing the workspace

VisualAge for Java gives you a range of characteristics that you can change to customize the IDE 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.

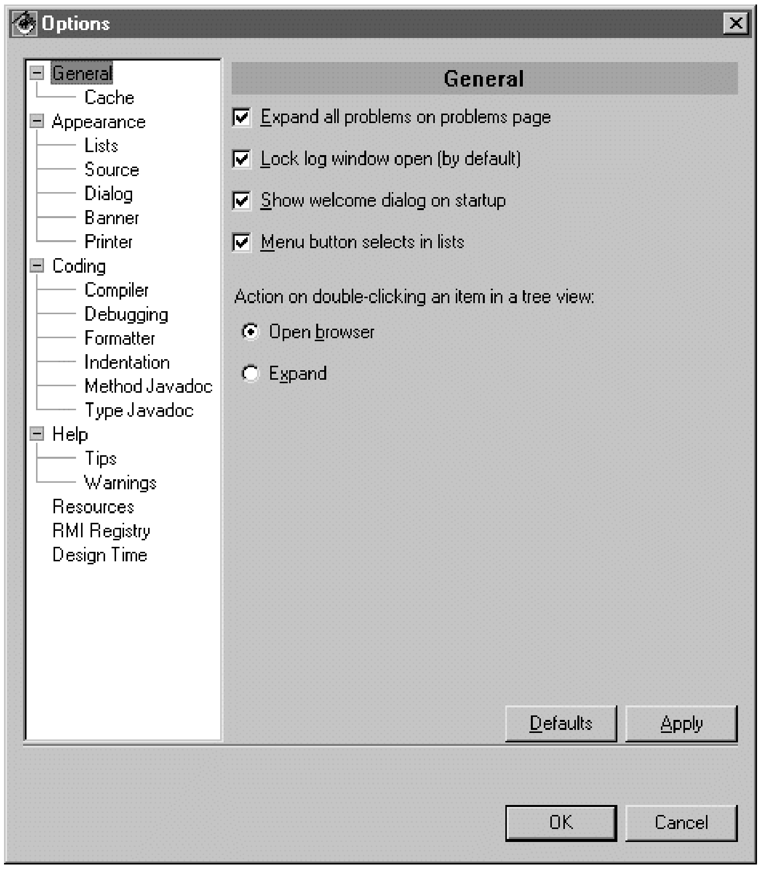
Setting customization options

In VisualAge for Java, you customize the workspace by setting options in the Options dialog. 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 opens the program element in a browser of its own. For example, when you double-click on a package icon, a package browser opens, showing all the types contained in the package. You can change this behavior so that double-clicking on a program element icon expands or collapses the program element tree beneath that icon.

To specify that double-clicking on a program element expands the tree view beneath the program element:

1. Select **Options** from the **Window** menu. The **Options** dialog appears.



☑ Expand all problems on problems page
☑ Lock log window open (by default)
☑ Show welcome dialog on startup
☑ Menu button selects in lists

Action on double-clicking an item in a tree view:
● Open browser
○ Expand

2. Select the **General** page, if it is not already selected.
3. Under **Action on double-clicking an item in a tree view**, enable the **Expand** radio button.
4. Click **OK**.

Now, when you double-click on a program element icon, the tree view expands to show the program elements contained by the clicked-on element.

To set all of the options on a page back to their default values, select **Defaults** and then select **OK**.

The tree view on the left-hand side of the Options dialog can be expanded to show all available pages.

Note: Each parent item in the tree (for example **Coding**) also has a page.

The following pages are available in the Options dialog:

- The **General** page displays options for some miscellaneous IDE behaviors. The **Cache** page displays options for memory management.
- The **Appearance** pages display options for how lists, source code, dialogs, banners, and printer output appear, including color, size, and font.
- The **Coding** page displays options for tabbing and saving code.
- The **Compiler** page displays options for reporting compiler errors.
- The **Debugging** page displays options for the integrated debugger and for generating a stack trace.
- The **Formatter** page displays options for the automatic code formatter, which standardizes how code is shown in Source panes.
- The **Indentation** page displays options for how each new line of code is indented.
- The **Method Javadoc** and **Type Javadoc** pages display options for generating standard Javadoc comments for each new method and type.
- The **Help** page displays options for specifying which web browser is used to display the help information.
- The **Tips** and **Warning** pages displays options for specifying which help tips and warnings you want to see.
- The **Resources** page lets you set the class path so that programs running in the workspace can find the resource files and classes that they need.
- The **RMI Registry** page displays options for remote method invocation (RMI).
- The **Design Time** page displays options for BeanInfo support and the Visual Composition Editor.

Domino AgentRunner

Lotus Domino is an application and messaging server with an integrated set of services that enable you to create interactive business solutions for the Internet and corporate intranets. The Domino AgentRunner is a tool which can be used to build, run, and debug Domino agents in VisualAge for Java.

The AgentRunner uses a set of debug classes that access Notes context information while you are running an agent in the IDE, on a Lotus Notes client.

To use the AgentRunner, you must follow these steps:

1. Set up your Notes and VisualAge for Java environments.
2. Create an agent in the IDE with a class that extends DebugAgentBase.
3. Export the class file into the file system.
4. Run the agent inside Notes to generate an AgentContext document in the AgentRunner.nsf. The AgentContext document is what allows you to build, debug and run without having to switch between the IDE and Notes.
5. Debug your agent in the IDE using VisualAge for Java's debugger.
6. Create the Production agent.

Each of these steps is described in more detail below.

Set up the AgentRunner

First you have to set up your Notes environment to access the AgentRunner classes, and then you have to set up the VisualAge for Java environment to access Domino Java classes and Notes AgentContext documents.

To set up your Notes 4.6 environment to support the AgentRunner, follow these steps:

1. Add the IVJAgentRunner.jar file to the JavaUserClasses statement in your notes.ini file. If you do not have a JavaUserClasses statement in your notes.ini file already, you can cut and paste the following statement to the end of your notes.ini file so that your JavaUserClasses points to the IVJAgentRunner.jar file.

   ```
   JavaUserClasses=X:\VAJava\ide\runtime\IVJAgentRunner.jar
   ```

   where `X:\VAJava` is the path where VisualAge for Java is installed. After you have edited the notes.ini file, you must shut down Notes and restart it so that your changes take effect.
2. Set your path to point to your Notes directory by entering

   ```
   set path=%path%X:/path/Notes
   ```

   on a command line. X is the drive on which Notes is installed. *path* is the path to your Notes directory. Your Notes directory will be temporarily added to your path. When you close your command window, this setting will be erased. Do not close the command window in which you set your path until you have finished using the AgentRunner. (If you want to permanently add this pointer, set the path to Notes in your computer's systems settings.)

3. Copy the AgentRunner.nsf file from X:\VAjava\ide\runtime (where `X:\VAjava` is the path where VisualAge for Java is installed) to your notes\data directory.

Your Notes environment is now set up to support the AgentRunner. Next, you have to set up your VisualAge for Java Environment.

Add the Domino Java classes to the Workbench

To set up your VisualAge for Java IDE to use the AgentRunner, add the Domino Java class library from the repository to your workspace. (From the Workbench window's **Selected** menu, select **Add** > **Project**. The Add Project SmartGuide will open.)

You now should see the Domino Java class library on the Projects page of the Workbench window. This project contains a package called lotus.notes, with all the Java classes for Notes Object Interface/Domino 4.6 and additional Debug classes that support the AgentRunner tool. You can now use these classes when running or debugging an agent in the IDE.

Import an agent from Notes

In the Workbench, create a project called Domino Agents.

Next, use the Import SmartGuide to import your agent from Notes into this new project. The imported Java code is compiled and any unresolved problems are added to the All Problems page. Your .java file appears as in a package in your Domino Agents project in the Workbench.

Create a new agent in VisualAge for Java:

Use the Create Class SmartGuide to create a class in your new package, in your Domino Agents project. Click on the class with mouse button 1. In the Source pane of the Workbench, write the code for your agent. (For instructions, see the *Java Programmer's Guide* for Lotus Notes).

Export the class file

When you have finished writing your agent, use the Export SmartGuide to save it and export the .class file from the IDE to the file system, so it can be read by Notes.

You can now generate your AgentContext document in a Notes Database.

Run the agent in Notes to generate an AgentContext document

Create a Java agent in a Notes database. See the Notes help for more information on creating an agent.

1. Open Lotus Notes 4.6.
2. Create your agent in the appropriate database.
3. Fill in the details for your agent. Select the **Java** radio button for **What should this agent run**
4. Click **Import class files** and select the file that you exported from VisualAge for Java.
5. Run the agent.

The AgentContext document is automatically generated in the AgentRunner.nsf when you run your agent in Notes from an agent class that extends DebugAgentBase. A call to getSession() will, after generating an AgentContext document, return null. So any use of the returned session will result in a thrown exception. But, since the purpose of running the agent is only to generate the context document (and not to run any of the agent code), you can ignore the exception.

Debug your agent

When you have generated an AgentContext document, you are ready to debug it in VisualAge for Java.

1. Set one or more breakpoints in the NotesMain() method of your agent.
2. Select your agent with mouse button 2.
3. Select **Tools - Domino AgentRunner.**

You have two options:

- Select **Properties** if you want to modify your AgentContext or select a different AgentContext. The AgentRunner window will open. Select the AgentContext that you wish to use and then click **Run Agent**. To modify your agent, click **Update Agent Context**. You can change the **Agent Runs on** and **Search Criteria** fields to generate the UnprocessedDocuments collection that you would like to use for debugging purposes. You must supply this information because it cannot be determined from running the agent. When you have finished updating your AgentContext, click the **Update AgentContext Document** button and close the window. Click **Run Agent**. (To make your new AgentContext the default choice, click **Save Selection** before you click **Run Agent**.)

or

- Select **Run** if you wish to run your agent with the default AgentContext document. The default is either the last AgentContext that you ran or the last one that you saved. If you have set any breakpoints or have errors in your code, the Debugger window will open and allow you to step through your code. See the online help for more information on using the debugger.

Create the production agent

When your development of the agent is complete, you can move your agent to Notes. First, in the IDE, you have to change your agent's base class to extend `lotus.notes.AgentBase` instead of `DebugAgentBase`.

Export the .class file to the file system. and reimport it in your agent in Notes.

You can now run your agent in Notes.

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 several 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
- **PDF Index -** provides a way of printing all information on a given topic.
- **Samples -** describes the samples that come with the product and how to get them. Some samples include directions on how to build them with the Visual Composition Editor.
- **Glossary -** defines terms used frequently in VisualAge for Java.

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.

To print all the information on a topic, use the PDF Index files.

Where you can get the latest VisualAge for Java information

To get the latest information updates, bookmark this Web site:

www.software.ibm.com/ad/vajava

The *Library* section provides additional Java programming books, papers and links.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

S430-4086-01