VisualAge for Java, Version 2.0

Visual Composition

Contents

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program 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. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, USA.

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 the 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.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

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

AIX
AS/400
DB2
CICS
IBM
OS/2

OS/390
RS/6000
San Francisco
VisualAge
Visual Servlet
WebSphere

Lotus, Lotus Notes and Domino are trademarks or registered trademarks of Lotus Development Corporation in the United States and/or in other countries.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation.

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

Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Chapter 1. Visual Programming Fundamentals

VisualAge for Java includes a state-of-the-art object-oriented visual composition editor for assembling program elements visually from JavaBeans components.

Object-oriented programming facilitates development of complex software systems by breaking them up into a number of much smaller, simpler program elements called *objects*. Objects work together by sending each other *messages*, that is, by requesting behavior that is implemented by the target object. Taken as a group, these behaviors comprise a *class interface*.

Using an object-oriented approach for complex systems provides the following benefits:

- Individual classes are much easier to create and understand.
- Systems are much easier to maintain and enhance. Object implementations can be modified individually without modifying the rest of the system, as long as the objects continue to respond appropriately to messages sent to them by other objects.

Despite these benefits, implementing large systems can still be expensive. One way to reduce the cost is to reuse object implementations. Many companies would prefer to buy reliable reusable classes, creating classes only for functions specific to their business. This vision of constructing custom software using standard building blocks has been called *construction from parts*. The building blocks themselves have popularly been called *parts* or *components*.

However, reuse is hard to achieve when the class interfaces are too specific to the application for which they were originally developed. To promote wider reuse, class interface conventions called *component models* have been defined, such as ActiveX, OpenDoc, and JavaBeans.

JavaBeans is the standard component model for the Java language and is the component model used by VisualAge. JavaBeans includes the following definitions:

An event model. Event models specify how a component sends messages to other objects without knowing the exact methods that the other object implements. This enables a component to be reused with a range of objects that have different interfaces

Events, properties, and methods. JavaBeans defines a component interface in terms of the events it can signal, the property values that can be read and set, and the methods it implements. This definition provides more structure to the interface of a component compared with a simple class interface, facilitating the use of tools such as the VisualAge Visual Composition Editor.

Introspection. Introspection refers to the ability to discover programmatically the component interface for instances of a particular component class. The reason to provide introspection is that it enables the use of programs such as the Visual Composition Editor that can work with component instances at run time without having the details of these components programmed into them.

The Visual Composition Editor enables you to create programs graphically from existing beans. Beans are simply Java classes that comply with the JavaBeans specification. JavaBeans is the component model supported and used throughout VisualAge, so this documentation will refer to VisualAge components as *beans*.

VisualAge provides user interface beans based on Java classes in the Abstract Windowing Toolkit (AWT) and Swing packages. The Visual Composition Editor is also extensible. It allows you to work with beans you create yourself, and it allows you to include beans imported into the environment from other sources. You can even create your own beans graphically using the Visual Composition Editor and then reuse these beans again within another program being created with the Visual Composition Editor.

To build a program with the Visual Composition Editor, you draw a picture using a canvas and palette of icons representing reusable beans. This picture specifies the set of beans that implements the function of the larger program (or bean) you are creating. For beans like user interface controls, the position of the controls relative to each other in the picture specifies how the controls will appear in the final program. For beans such as database components, the position of the bean in the picture generally has no significance.

The Visual Composition Editor provides a very sophisticated connection capability to specify how components of the picture will interact to implement functions of the program. Using connections, much of the behavior of an application can be specified graphically. Connections also allow you to integrate custom code written in the Java language.

See the JavaBeans Home Page for links to detailed information on JavaBeans and BeanInfo.

RELATED CONCEPTS

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 5. Bean Design for Visual Composition” on page 19

Chapter 2. How Classes and Beans Are Related

VisualAge beans are Java classes that conform to the JavaBeans component architecture. *Composite* beans are made up of embedded beans. We use the term *bean* to refer to both a class and its instances, as follows:

- When we refer to beans on a palette or to beans that you create by writing code, we mean bean **classes**.
- When we refer to beans on the free-form surface or to beans that are connected, we mean bean **instances**. Sometimes, beans represent instances that have not yet been created.

During visual composition, you interact with bean interfaces. The most useful bean interfaces contain the following features:

Access to data, or ***properties*****.** A complete property interface includes methods to return the value of the property, to set the value of the property, and to notify other beans when the value of the property changes. The interface for a property does not have to be complete. For example, a property might be read-only, in which case the interface would not support the ability to set the value of a property. A property can be any of the following:

- An actual data object stored within the bean, such as the street in an address bean
- A computed data, such as the sum of all numbers in an array or the profit that is computed by subtracting dealer cost from the retail price

Access to the behavior of a bean, or ***methods*****.** These represent tasks you can ask a bean to perform, such as open a window or add an object to a collection of objects.

Event notification. By signaling events, a bean can notify other beans that its state has changed. For example, a push button can signal an event to notify other objects when it is clicked, or a window can signal an event when it is opened, or a bank account can signal an event when the balance becomes negative.

Events can also be signaled when the value of a bean property changes, such as when money is deposited into or withdrawn from a bank account. In this case, the balance property is said to be *bound* to an event.

RELATED CONCEPTS

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 5. Bean Design for Visual Composition” on page 19

RELATED REFERENCES

“Chapter 37. Factory and Variable Beans” on page 189

Chapter 3. Visual, Nonvisual, and Composite Beans

You can use many kinds of beans to construct program elements. All beans exist as either primitives or composites. Primitive beans are the basic building blocks from which composites are constructed. You must construct new primitive beans using a programming language because there are no similar beans to use in building them. Primitive beans can be either visual or nonvisual.

Visual beans are elements of the program that the user can see at run time. The development-time representations of visual beans in the Visual Composition Editor closely match their runtime visual forms. Users can edit these beans in the Visual Composition Editor in their visual runtime forms. Examples include windows, entry fields, and push buttons. In general, visual beans are subclasses of *java.awt.Component*.

Nonvisual beans are elements of the program that are not necessarily seen by the user at run time. On the Visual Composition Editor's free-form surface, users can manipulate these beans only as icons. Examples include business logic, database queries, and communication access protocol beans.

Beans that have a visual representation at run time but do not support visual editing are treated as nonvisual. Examples of this kind of nonvisual bean include message boxes and file selection dialogs.

Composite beans can contain both visual and nonvisual components. In general, composite beans are based on one of these classes, but you are by no means limited to these:

- *Applet* or *JApplet*, for Web applets
- *Frame* or *JFrame*, for GUI applications
- *Panel* or *JPanel*, for reusable GUI surfaces embedded in either applets or applications
- *VisualServlet*, for servlets

RELATED CONCEPTS

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 4. Visual Composition Editor Overview” on page 7

RELATED TASKS

“Chapter 26. Incorporating User-Written Code into Visual Composites” on page 129

Chapter 4. Visual Composition Editor Overview

Visual composition is the creation of object-oriented program elements by manipulating graphical representations of components. VisualAge provides a powerful tool, the Visual Composition Editor, that enables you to visually construct applications, applets, and reusable beans.

In the Visual Composition Editor, you select and place beans to create graphical user interfaces (GUIs). These GUIs can include VisualAge beans, imported beans, and beans you create yourself. By following a few guidelines, you can design versatile beans that you can reuse in many compositions. VisualAge also enables you to use nonvisual beans to perform the business logic and data access.

Development using visual composition can include the following steps:

1. Design your program elements. Determine what you can compose visually and what you must write by hand.
2. Create nonvisual beans.
3. Using the Visual Composition Editor, enhance these classes by dropping beans and setting initial values for properties. Extend the behavior of VisualAge beans by writing code.
4. For business logic, add code to the appropriate class stubs.
5. Connect beans to define the program element’s behavior and flow.
6. Save your work. VisualAge generates and compiles the code for visually composed beans. Select 🏃 **Run** in the Visual Composition Editor to try out the finished product.

RELATED CONCEPTS

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

“Free-Form Surface” on page 8

“Beans Palette” on page 9

“Adding Beans in the Visual Composition Editor” on page 10

“Property Sheets” on page 11

“Setting Tabbing Order” on page 12

“Tearing Off Properties” on page 13

“Layout Managers in Visual Composition” on page 14

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project

“Setting a Layout Manager during Visual Composition” on page 60

“Chapter 21. Managing the Beans Palette” on page 85

“Setting the Tabbing Order” on page 54

“Tearing Off Properties” on page 56

“Opening the Property Sheet for a Bean” on page 56

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Properties” on page 196

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Free-Form Surface

The *free-form surface* is the large open area in the Visual Composition Editor. It is like a blank sheet of paper or a work area where you can add, manipulate, and connect the beans that you use to create your composite bean.

Some of the functions you can perform on the free-form surface include:

- Add visual beans.
- Add nonvisual beans to build the application logic for a composite bean.
- Delete beans.
- Connect beans to define behavior.

You cannot edit on the free-form surface if the bean you are attempting to edit meets any of the following conditions:

- The class is in a system package.
- An exception occurred during the creation of the bean.
- The class is a Java AWT lightweight component.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Properties” on page 196

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Beans Palette

The *beans palette*, located on the left side of the Visual Composition Editor, provides building blocks you can use to construct a program element. It consists of categories in a drop-down list, each one containing groups of beans.

You can add a bean to your program element by first selecting the category, then a bean, and then dropping it on the free-form surface, a container bean, or the beans list.

To add multiple instances of the same bean, enable **Sticky** by holding Ctrl while selecting the bean. Selecting a new bean or the Selection tool disables **Sticky**.

The status area displays the name of the selected category and bean.

The beans palette initially contains the following:

- Categories (including **AWT**, **Swing**, and **Other**) and the option to load **Available** installed features to the workspace image.
- Beans provided with VisualAge
- “Selection Tool”
- “Choose Bean Tool” on page 10
- A pop-up menu

You can modify the palette by resizing it, changing the icon size, adding categories, adding separators, adding beans you have constructed yourself or beans supplied by a vendor, or removing separators, categories, or beans. In addition, the **Available** feature in the category menu button, loads installed features to the workspace. This may include the addition of categories and beans to the palette. Modifying the palette can increase your productivity in the following ways:

- Reduce the time and effort required to place beans that you have created and that you use often in the Visual Composition Editor.
- Reduce the time and effort required to place vendor beans or beans from another project.
- Eliminate the need for manually placing beans through the Choose Bean tool, which requires that you use the exact class name of the bean.

When you add a new bean to the palette, the entire visual or nonvisual bean is represented with the default puzzle icon unless you designate another icon in the BeanInfo Class. Once you add beans to the palette, you can place them in the free-form surfaceVisual Composition Editor in the same way you place beans that VisualAge provides.

Selection Tool

Select the **Selection tool** to unload the mouse pointer and return it to the selection pointer. The loaded mouse pointer appears as a crosshair and carries a bean that can be added to the free-form surface, the beans list, or to an existing container bean. When unloaded, the mouse pointer reverts to an arrow that you use to select and perform actions on beans. If the mouse pointer is not loaded, this tool is not enabled..

Choose Bean Tool

Select the **Choose Bean** tool to retrieve a bean that is not on the palette and drop it on the beans list, free-form surface, or an existing container bean.

RELATED TASKS

“Chapter 21. Managing the Beans Palette” on page 85

RELATED REFERENCES

“Chapter 42. Modify Palette Window” on page 209

“Modify Palette” on page 204

“Selection Tool” on page 9

“Chapter 43. Choose Bean Window” on page 211

Adding Beans in the Visual Composition Editor

When you place beans in the Visual Composition Editor:

- Avoid overlaying beans

  It is not good interface design for one bean to overlay another bean. Completely or partially overlaying a bean can result in focus problems, causing users to see but be unable to select the bean.

- Embed composite beans into other composites

  By embedding composite beans into other composites, you minimize the confusing spider effect of connection lines. For example, you can create a composite bean that consists of a panel on which you have placed buttons and check boxes, and make connections. When you embed this bean in your main interface, you cannot see the connection lines. You place and work with the composite as one bean—not as a panel and separate buttons and check boxes.

  If you need to edit the composite or its internal connections, you simply select **Open** from the pop-up and the Visual Composition Editor for the composite appears, as described in “Editing Beans within a Composite Bean” on page 53.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

"Properties" on page 196

"Chapter 41. Pop-Up Menus for the Visual Composition Editor" on page 201

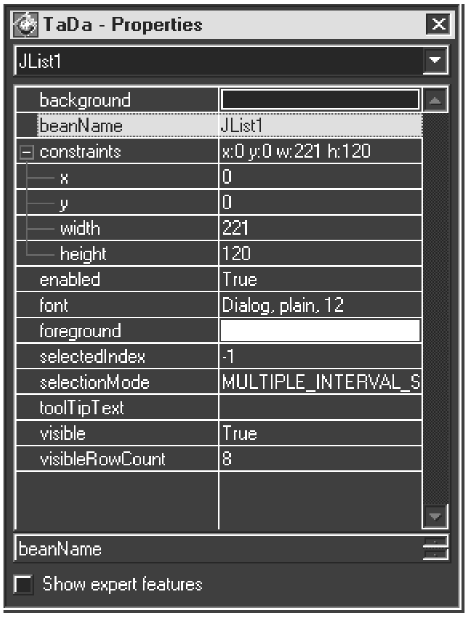
Property Sheets

The bean property sheet provides options for changing the initial appearance and state of beans. You can open the property sheet from the pop-up menu of a selected bean either in the Visual Composition Editor or Beans List window. You can also select Properties from the **Tools** pull-down menu or select ▦ from the tool bar.

The left column of the bean property sheet contains a list of properties and the right column contains the editable values. An expansion icon ⊞ to the left of the property indicates that the property has more editable values. For example, when you expand the constraints, sizing properties, such as x, y, width, and height, appear for editing. When you select the value column of a property, you are provided with an editing option. For example, if you want to modify the property for a label, select the value field for *label* and enter the new label in the entry field. If you want to change the background color for the same bean, a small button ... appears when you select the value column for *background*. When you select this button, a dialog window appears with color options.

You can specify the type of property editor to associate with the property by setting the propertyEditor field in the BeanInfo. For more information, see ″Enabling Custom Edit Support for Your Bean.″

Once you open a property sheet, you can modify properties for most beans in Visual Composition Editor. To edit another bean select it in the Visual Composition Editor or from within the property sheet by selecting the bean from the drop-down list at the top of the property sheet. If you open a property sheet after selecting multiple beans, the property sheet provides editing options for only the common values of the selected beans. For example, you can use this feature to set the left and top inset of several beans in a GridBagLayout at once.



TaDa - Properties

JList1

background	■
beanName	JList1
⊟ constraints	x:0 y:0 w:221 h:120
x	0
y	0
width	221
height	120
enabled	True
font	Dialog, plain, 12
foreground	□
selectedIndex	-1
selectionMode	MULTIPLE_INTERVAL_S
toolTipText	
visible	True
visibleRowCount	8

beanName

□ Show expert features

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project
“Opening the Property Sheet for a Bean” on page 56

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Properties” on page 196

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Setting Tabbing Order

The tabbing order is the order in which the input focus moves from bean to bean as the user presses the Tab key. The initial tabbing order is determined by the order in which you drop the beans. The first bean in the tabbing order receives the initial input focus. For example, if the first bean in the tabbing order is a button, that button receives the initial input focus when the application starts.

The tabbing order can be set or displayed only for beans that are placed within a composite bean. For example, if you place a row of buttons in a frame window, you can set the tabbing order for the buttons.

If the tabbing order includes each entry field in which a user can type, the user can move the input focus from one entry field to another. Arrow keys only move the cursor within an entry field; only the Tab key, backtab key, and mouse can change the input focus from one entry field to another. Read-only fields do not need to be included in the tabbing order.

Because the order in which beans are placed on a composite bean determines the tabbing order, you will probably need to change the order as you add or rearrange beans.

For example, drop three buttons and then rearrange them so that Button3 is between Button1 and Button2. The tabbing order of these buttons is Button1, Button2, Button3, even though Button3 is now between Button1 and Button2. You must change the order to have the focus move from Button1, Button3, and Button2.

The color of the tab tags reflect information about the beans that you use. Yellow tab tags represent simple beans, such as entry fields and buttons. Blue tab tags represent composite beans and container beans.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project
“Setting the Tabbing Order” on page 54

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Properties” on page 196

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Tearing Off Properties

You tear off a property to gain access to the encapsulated features of a bean. This can be necessary when a property is in itself a bean and you want to connect to one of its features. The torn-off property is not actually a separate bean but a variable that represents the property itself or points to it.

For example, in an address book application you might tear off properties as follows:

- You might have a Person bean that contains both *homeAddress* and *workAddress* properties, both of which, in turn, could contain *street*, *city*, and *state* properties.
- Tearing off a *homeAddress* or *workAddress* property makes the nested *street*, *city*, and *state* properties directly accessible. Now that the nested properties are directly accessible, you can make connections to and from them, as well as to their associated events and methods.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project
“Tearing Off Properties” on page 56

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Properties” on page 196

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Layout Managers in Visual Composition

Many container components support the use of *layout managers*. A layout manager is a class that implements the *java.awt.LayoutManager* or *java.awt.LayoutManager2* interface.

You assign a layout manager to the container. In most layouts, you can then define properties for the layout that govern the specifics of the sizing and resizing behavior for the components.

Visual composition makes it easy to try different layouts. If you prefer to lay beans out individually, you can use the null layout setting (that is, no layout) and the Visual Composition Editor alignment tools. For examples of using layout managers, see the *com.ibm.ivj.examples.vc.layoutmanagers* classes shipped in the IBM Java Examples project.

Supported Layouts

VisualAge supports the use of layout managers in container beans, as follows. For several of these layouts, VisualAge sets layout properties by default.

- BorderLayout arranges components along each edge of the container (North, South, East, and West), and one component in the center.

If the container bean uses a layout other than border and has more than 5 components, changing the container layout to border could result in overlaid components. The beans list allows you to easily perform tasks on the covered components.

- BoxLayout arranges components vertically or horizontally without wrapping. You can nest multiple panels with different combinations of horizontal and vertical BoxLayout to achieve an effect similar to GridBagLayout, without the complexity. If you are using the vertical alignment, BoxLayout attempts to make all components the same width. With horizontal alignment, BoxLayout attempts to match component height.
- CardLayout arranges components in a linear depth sequence similar to a deck of cards, notebook, or tabbed dialog box. Each component is called a *card*.

  You can use **Switch To** on the pop-up menu to move through the deck, or perform tasks on the covered cards in the beans list.
- FlowLayout arranges components in horizontal lines. The alignment property in the layout manager enables you to specify where you want the flow to begin.
- GridLayout arranges components in a table, all cells having the same size.
- GridBagLayout enables you to arrange components in a highly complex grid. As you add or move beans, the free space shuffles so that beans are centered on the interface, while retaining your arrangement. Grid cells are not necessarily identical in size and components can span multiple cells. You can customize grid sizing behavior down to each individual component.
- Null layout means that no layout manager is assigned. Without a layout manager, resizing the container at run time does not affect the size and position of the components.

  You can customize components within the null layout by means of dragging the beans, using Tool Bar options, or through the **constraints** option in Properties.

Alignment tools are disabled for all but null layout.

For examples of using layout managers, see the *com.ibm.ivj.examples.vc.layoutmanagers* classes shipped in the IBM Java Examples project.

Setting Layout Properties during Visual Composition

- BorderLayout—you can specify the spacing between adjacent components.
- BoxLayout—you can specify whether components are layout out vertically or horizontally
- CardLayout—you can specify the spacing between adjacent components.
- FlowLayout—you can specify the spacing between adjacent components, and the alignment to start at center, left, or right.
- GridBagLayout—you do not specify additional layout properties for a container that uses GridBagLayout. However, you can specify constraints for the components within the container.
- GridLayout—you can specify the spacing between adjacent components, and the number of rows and columns.

For layout manager details, see the Java API documentation.

Consider waiting to set layout properties until you have settled on a layout manager. Many values are lost when you switch layouts or move the component to another container on the free-form surface.

Dropping Beans into the Layout

Once you have assigned a layout, the Visual Composition Editor provides visual cues to help you place beans in the correct position. These cues appear when you place the loaded mouse pointer in position and then press and hold mouse button 1:

- For FlowLayout and GridLayout, a bold vertical bar appears. If you release the mouse, VisualAge places the bean to the right of the vertical bar.
- For BorderLayout and null layouts, an outline of the placement options for the bean appears: for BorderLayout the regions (North, South, etc.), for null layout the container. When you release the mouse, VisualAge places the bean within the outline under the crosshair.
- For GridBagLayout, a bold grid appears that is based on the beans dropped so far. VisualAge attempts to place the bean as indicated by the pointer. If the pointed-to cell is empty, all borders of the cell are highlighted. If the pointer rests on a row or column boundary, a new row or column is inserted. New rows appear below the pointer; new columns appear to the right.

For CardLayout, the container is outlined. VisualAge adds beans to the top of the card deck, making the first bean you dropped the bottom card. You can use **Switch To** on the pop-up menu to move through the deck, or perform tasks on the covered cards in the beans list.

To get access to the layout interface directly, drop a Variable bean on the free-form surface to the right of the container. Change the type of the Variable bean to that of the class implementing the layout manager interface (for example, CardLayout). Connect the *layout* property of the container bean to the *this* property of the Variable bean. Then connect to features of the Variable bean.

If you use a layout that allows for a bean to completely cover another bean, the beans list enables you to easily perform tasks on the covered components. To modify bean placement on the Visual Composition Editor from within the beans list, open the Properties for the bean and modify the Constraints.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 17. Composing Beans Visually” on page 67

Adding the IBM Java Examples project

“Setting a Layout Manager during Visual Composition” on page 60

“Setting Layout Properties during Visual Composition” on page 60

“GridBag Layout Constraints” on page 60

“Creating a GUI Using GridBagLayout” on page 61

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Properties” on page 196

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Chapter 5. Bean Design for Visual Composition

Designing a good bean is very similar to designing a good class: it must be usable. In fact, making a bean usable for visual composition improves its use in handcoding as well. Consider the following:

- Implement a null constructor.
- Keep your bean small. Minimize dependencies on other beans and classes.
- Implement the *java.io.Serializable* interface.
- For visual beans, subclass from *java.awt.Component* or one of its subclasses.
- Make important functions available through settable properties. If necessary, provide a custom editor to make setting properties easier. Avoid dependencies upon the order in which properties are set.
- Mark interface features in BeanInfo to optimize clarity:
  - Set *preferred* to `true` for those features that most people will use.
  - Set *expert* to `true` for those properties that most people will never use.
  - Set *hidden* to `true` for those properties that must not be used in connections.
  - Set *Design time property* to `false` for those properties that you do not want surfaced in the beans property sheet.
- To take advantage of reflection, follow standard design patterns for methods, events, and properties. Do not use the same set method names for different properties. Provide a BeanInfo class with meaningful display names and descriptions.
- Set up bound properties where appropriate. However, be careful not to overdo it, because property events are multicast through PropertyChangeEvent, which can affect performance.
- Have your bean signal events for significant state changes. Use unique event classes instead of a single event class with a flag in eventData.
- Provide .gif files so that the bean can be represented in the Visual Composition Editor. Include both 16x16 and 32x32 versions, with transparent backgrounds.
- To minimize the number of connections during visual composition, specify several methods with a small number of parameters rather than a single method with many parameters.

RELATED CONCEPTS

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 6. Use of Visual Beans Created in Other Tools” on page 21

RELATED TASKS

“Chapter 27. Defining Bean Interfaces for Visual Composition” on page 131

“Chapter 23. Enabling Custom Edit Support for Your Bean” on page 117

RELATED REFERENCES

JavaBeans specification

Chapter 6. Use of Visual Beans Created in Other Tools

VisualAge can generate visual composites from GUI beans created outside the VisualAge development environment. This can save you editing time if the bean contains a large number of visual components. VisualAge constructs the composite by querying an instance of your GUI bean about its contents. This is what you can expect from this reverse-engineering process:

- VisualAge constructs only those controls that ultimately inherit from *java.awt.Component*.
- Each component must have a null constructor; VisualAge writes over the null constructor of the class being edited to make it consistent with that for other visual components. Be sure that all visuals embedded within a component also have null constructors.
- VisualAge also generates its own names for visual components, so any handwritten code referring to the previous component names must be updated with the new names. If you want VisualAge to use the names you assigned in the original bean, explicitly set the name of each component using the *setName( )* method before you proceed with reverse-engineering. An example follows:

  ```
  if (myBean == null) {
     myBean = new JTextField();
     myBean.setName("myBean");
  }
  ```
- VisualAge writes over methods and fields whose names collide with those it typically generates. This most commonly occurs with get and set accessor methods. For a complete list, read “Chapter 9. Generated Code” on page 33.
- Layout and property settings for visual components are preserved whenever possible. Custom layout managers are not supported.
- No attempt is made to construct connections from method calls, but you can draw new connections as soon as VisualAge has constructed the composite. In most cases, you will have to draw event connections to get the bean to behave as it did before the reverse-engineering.
- Embedded composites are constructed as primitives. Reverse-engineer the embedded composites first.
- Serialized instances are constructed from the class, not from a serialization file. As with nonserialized components, VisualAge sets properties from BeanInfo queries.
- When VisualAge does not have enough information to set a property, the property is not set. (One example of this is the *icon* property of JButton and other Swing components.)

Undo is not supported for reverse-engineering visual composites. If you are not satisfied with the results, close the class browser without saving the newly engineered composite. In any case, version the bean before you reverse-engineer it.

To proceed with reverse-engineering, open the bean in the Visual Composition Editor. From the **Bean** menu, select **Construct Visuals from Source**.

RELATED CONCEPTS

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

RELATED TASKS

Chapter 7. Bean Interfaces and BeanInfo

The bean interface defines the property, event, and method features of your bean. These features can be used in visual composition when your bean is added to another bean. A BeanInfo class describes the bean and features that you add to the bean. Other features are inherited from the superclass of your bean unless you choose not to inherit features.

You can define a bean interface in the following ways:

- In the Workbench window, create a new bean class based on a class with features you need. By default, the new bean inherits the features of the class it extends. You can control feature inheritance by setting the **Inherit BeanInfo of bean superclass** option in the **Design Time** pane of the Options window. Open the Options window from the **Window** menu of the Workbench.
- In the BeanInfo page, add new features to a bean. You can add features to extend the inherited feature set, to override inherited features, or both. When you add a feature to a bean, VisualAge generates code that describes the feature in the BeanInfo class for the bean.
- In the Visual Composition Editor, promote features of embedded beans to the interface of a composite bean. When you promote a feature of an embedded bean, VisualAge generates code that describes the promoted feature in the BeanInfo class for the composite bean.

When you create a new bean, it does not initially have a BeanInfo class. VisualAge creates a BeanInfo class when you add or promote the first feature that is not inherited, or when you explicitly direct VisualAge to create a BeanInfo class. You can create a BeanInfo class in the BeanInfo page.

RELATED CONCEPTS

"Promotion of Bean Features"

"Default Promoted Feature Names" on page 24

"Feature Naming Guidelines" on page 24

"Chapter 2. How Classes and Beans Are Related" on page 3

"Chapter 5. Bean Design for Visual Composition" on page 19

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

Promotion of Bean Features

When you create a composite bean, you might want some features of beans that are embedded within it to appear in the interface of the composite bean. For example, suppose you create a composite bean named ButtonSet containing a set of buttons that you want to reuse. When you add the ButtonSet composite bean to another composite, you want to be able to connect to each of the buttons.

To add features of embedded beans to the interface of a composite bean, you must promote them to the composite's interface. To add an entire embedded bean as a property of the composite bean, promote the *this* property of the embedded bean.

When you promote a feature of an embedded bean, VisualAge generates code that describes the promoted feature in the BeanInfo class for the composite bean. After the feature is promoted, you can manage the feature in the BeanInfo page the same as you manage features that you add there.

When you add a bean with promoted features to another bean, you can use the promoted features the same as you use other features of the bean. If you add a bean that has an embedded bean as a property, you can tear off the property as a Variable. Then, you can access the features of the embedded bean referenced by the Variable.

RELATED CONCEPTS

“Chapter 7. Bean Interfaces and BeanInfo” on page 23

“Default Promoted Feature Names”

“Tearing Off Properties” on page 13

RELATED TASKS

“Promoting Features of Embedded Beans” on page 137

Default Promoted Feature Names

When you promote a feature of an embedded bean in the Promote Features window, you can use a default feature name produced by VisualAge. If you do not want to use the default feature name, you can change the name.

The default feature name is a combination of the name of the embedded bean and the name of the feature you are promoting. This identifies the bean that implements the feature, which is helpful if the composite bean contains more than one bean with the same feature. Then, when you connect to the feature, you can tell which embedded bean it belongs to.

For example, if you promote the *enabled* property for a bean named YesButton, the default composite bean feature name is *yesButtonEnabled)*.

RELATED CONCEPTS

“Chapter 7. Bean Interfaces and BeanInfo” on page 23

“Promotion of Bean Features” on page 23

RELATED TASKS

“Promoting Features of Embedded Beans” on page 137

Feature Naming Guidelines

When you add or promote a feature, use these naming guidelines:

- Begin the feature name with a lowercase character. Features represent methods, which typically have names that begin with a lowercase character. By contrast, class names typically begin with an uppercase character.
- Do not use blanks in the feature name.

RELATED CONCEPTS

“Chapter 7. Bean Interfaces and BeanInfo” on page 23

RELATED TASKS

"Adding Property Features" on page 133

"Adding Method Features" on page 134

"Adding Event Features" on page 135

"Promoting Features of Embedded Beans" on page 137

Chapter 8. Connections

When you make a connection in the Visual Composition Editor, you define the interaction between components. For example, if you want a data value to change when an event occurs, you would make an event-to-property connection. The following table summarizes the types of connections that the Visual Composition Editor provides. The return value is supplied by the connection's *normalResult* event.

Table 1. Connection Type Summary

If you want to...	**Use this connection type**	**Color**	**Does connection have a return value?**
Cause one data value to change another	property-to-property	Dark blue	No
Change a data value whenever an event occurs	event-to-method	Dark green	Yes
Call a public behavior whenever an event occurs	event-to-method	Dark green	Yes
Call a behavior whenever an event occurs	event-to-code	Dark green	Yes
Supply a value to a parameter	parameter-from-property, parameter-from-code, or parameter-from-method	Violet	No

The Source and Target of a Connection

A connection is directional; it has a source and a target. The direction in which you draw the connection determines the source and target. The bean on which the connection begins is the *source*; the bean on which it ends is the *target*.

Often, it does not matter which bean you choose as the source or target, but there are connections where direction is important.

- In an event connection, the event is always the source.
- For property-to-property connections, if only one of the properties has a public set method, VisualAge makes that property the target. This is done so that the property that has the public set method can be initialized at run time.
- When you make property-to-property connections, the order in which you choose the source and target is important. The source and target property values may be different when the bean is first initialized. If they are, VisualAge resolves the difference by changing the value of the target to match that of the source if the properties are bound. Thereafter, if both properties have public set methods, the connection updates either property if the other changes.

The target of a connection can have a return value. If it does, you can treat the return value as a feature of the connection and use it as the source of another connection. This return value appears in the connection menu for the connection as *normalResult*.

Property-to-Property Connections

A *property-to-property connection* links two property values together. This causes the value of one property to change when the value of the other changes, except as noted in the table below. A connection of this type appears as a bidirectional dark blue line ○———● with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source.

When your bean is constructed at run time, the target property is set to the value of the source property. These connections never take parameters.

For indexed properties,VisualAge generates two get/set method pairs—one for the array and one for accessing elements within the array. When you connect indexed properties, VisualAge uses the accessors for the entire array. If you want to access an individual element, make a method connection to the specific accessor.

To achieve the behavior that you anticipate, you must know something about the properties you are connecting. The following table shows the results of connecting properties of different types.

Table 2. Behavioral Considerations for Connections

	TARGET HAS...		
SOURCE HAS...	Set method and Event	Event only	Set method only
Set method and Event	Source and target values are fully synchronized.	This connection is not valid.	The source initializes the target. The target updates whenever the source’s value changes.
Event only	The source initializes the target. The target updates whenever the source’s value changes.	This connection is not valid.	The source initializes the target. The target updates whenever the source’s value changes.
Set method only	The source initializes the target only. The source updates whenever the target’s value changes.	This connection is not valid.	The source initializes the target. No further updates occur.

A *bound* property is a property whose value changes as the result of an event. For example, when a deposit or withdrawal event occurs from a bank account, the balance property is said to be *bound* to the event.

Properties in a property-to-property connection do not synchronize if:

- The property is not bound and has no associated event.
- At least one property is constrained, or prevented from changing under certain conditions, and a change is vetoed.
- The event to which the properties are bound is asynchronous. See the bean information for details on selecting another event.

Because most of the properties in this version of AWT are not bound, the source initializes the target when constructed and performs no further updates. If you want the property values to synchronize when an event occurs, you must associate an event with the property. You can do this by opening the property sheet for that connection and selecting an event from the source or target event fields. For example, the text property in AWT components is unbound. You can force this property to fire by selecting the textChanged event from the event field.

Event-to-Property Connections

An *event-to-property* connection updates the target property whenever the source event occurs. An event-to-property connection appears as a unidirectional dark green arrow → with the arrowhead pointing to the target.

The property must have a public set method known to VisualAge; otherwise, you cannot make the connection. If you open properties on a connection of this type, the target of the connection appears as a method with the same name as that of the target property.

Event-to-Method Connections

An *event-to-method* connection calls the specified method of the target object whenever the source event occurs. An event-to-method connection appears as a unidirectional dark green arrow → with the arrowhead pointing to the target.

Often much of an application’s behavior can be specified visually by causing a method of one bean to be invoked whenever an event is signalled by another bean. For example, you might invoke the dispose method on a Frame bean when the actionPerformed event is signalled by a button (this happens when the user clicks the button).

A connection with a dashed line requires parameters. You can provide parameters through a parameter connection, by passing event data (an option in the connection window), or through a return value. For more information, see the Task information on connections.

To access behavior that is not part of the bean interface, use code connections.

Code Connections

A *code* connection calls code of the composite bean whenever the source event occurs. This type of connection appears as a unidirectional dark green arrow → with the arrowhead pointing to a moveable text box on the free-form surface.

If you want processing to occur when a bean in your composition signals an event, but no available bean has a public method to accomplish that process, you can write and connect to a custom private method of the class you are editing. These methods are called code to distinguish them from the public methods for the composite class that you create and publish as bean methods. Technically, your code does not need to be private, and it is not different from other Java methods.

Note: You might notice that in VisualAge Java, the word *method* is used in two subtly different ways. In the Java language, *method* refers to a callable function of any class. In JavaBeans, *method* refers to a subset of the Java class methods that are exported as bean features. The set of JavaBeans method features is often the same as the set of Java public methods. However, the bean provider may further restrict the set of Java methods that show up as JavaBeans features.

You can use a code connection for the bean you are developing to connect to:

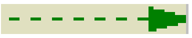
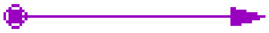
- Private methods
- Public methods
- Protected methods
- Package-private methods
- Public methods of any of its embedded beans

You cannot, however, connect to private or protected methods of embedded beans.

You might want to create a code connection to:

- Reduced the number of connections in a bean.
- Encapsulate repeated tasks that are specific to the bean being developed.
- Keep an operation internal to the class, such as a composite bean performing a calculation the user does not need to be aware of whenever a value changes.

Parameter Connections

In most cases, when a connection needs a parameter, the connection line appears dashed.  A parameter connection supplies an input value to the target of a connection by passing either the value from a property or the return value from a method. In a parameter-from-method connection, the connection appears as a unidirectional violet arrow  with the arrowhead pointing from the parameter of the original connection to the method providing the value. In a parameter-from-property connection, the connection appears as a bidirectional violet line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source.

The original connection is always the source of a parameter connection; the source feature is the parameter itself. If you select the parameter as the target, VisualAge reverses the direction of the parameter connection automatically.

If the target of the original connection takes parameters and the same event provides parameters by default, the connection line might appear solid. This is true even if the target takes one input parameter and you have not otherwise provided one. VisualAge can use any of the following means to supply parameters with values:

- If the parameter is connected to a property, the connection calls the get method for the property to get the value for the property and return it to the parameter.
- If the parameter is connected to a method, the connection code calls the method and passes the return value for the method to the parameter.
- If the source of the original connection passes event data in the connection code, VisualAge applies it to the parameter. If several values are required, event data is applied to the first parameter only.

- If you specify a constant parameter value in the original connection, VisualAge passes it in the connection code.

RELATED CONCEPTS

"Generated Feature Code" on page 35

RELATED TASKS

"Chapter 19. Connecting Beans" on page 77

"Changing the Source and Target of Connections" on page 83

"Supplying a Parameter Value Using a Constant" on page 79

RELATED REFERENCES

"Chapter 28. Beans for Visual Composition" on page 141

"Chapter 47. Property-to-Property Connection Window" on page 219

"Chapter 48. Event-to-Method Connection Window" on page 221

"Chapter 50. Event-to-Code Connection Window" on page 225

"Chapter 49. Constant Parameter Value Properties Window" on page 223

"Chapter 51. Parameter-from-Code Connection Window" on page 227

Chapter 9. Generated Code

With the Visual Composition Editor, you lay out beans graphically and specify their interaction using a high-level connection model. When you save your composition, VisualAge generates Java code that maps this graphical representation to the JavaBeans component model and to the APIs for the beans themselves. The resulting generated code is best understood if you are familiar with the JavaBeans component model; in fact, it is quite similar to code that an expert JavaBeans programmer would have written by hand. If you are interested in understanding more about the code generated by VisualAge, read the JavaBeans specification first; then read the rest of this section and look at the code generated by VisualAge for different visual compositions.

Some items are generated only once; others are regenerated every time the product detects a relevant change. Some items are generated only if you have indicated to VisualAge that you want certain capabilities present in your bean. Methods that are regenerated whenever the product detects a relevant change are indicated by to the right of the method signature.

Note: The code resides in the VisualAge repository, not as code files in your working directory. To get a copy of your code or compiled class in a file, you must export the class.

Code Generated from Visually Composed Beans

When you compose your bean visually, VisualAge generates much of the user interface code for you. The connections you make between beans are often sufficient to define behavior at run time. If not, you can extend it by adding your own code.

In addition to other items generated for all classes, VisualAge generates the following code for a composite bean. This code is generated when you save the bean, select **Re-generate Code** from the Bean menu of the Visual Composition Editor, or select **Run**:

- A field declaration for each embedded bean, which takes its name from the name given to the bean when it was dropped in the Visual Composition Editor. This field is declared as private. To minimize potential collisions between handwritten and generated code, all generated field names start with *ivj*.
- A private get method for each embedded bean, which takes its name from the name given to the bean when it was dropped in the Visual Composition Editor. Because Variable and Factory beans represent instances that do not exist at initialization, VisualAge generates public get and set methods for them so that they can be set at run time. Serialized instances are restored in the get method through a call to *Beans.instantiate( )*; other instances are created in the get method in a `new` expression if they do not already exist.

  Note: Take care when renaming embedded beans or torn-off properties. Otherwise, VisualAge might generate an accessor method that overrides an inherited method or overwrites a user-written method.

- For each promoted feature, a public get and set method. If the promoted feature's name is the same as that of the bean in which the feature resides, VisualAge generates the bean's get and set methods as public instead of private.
- If at least one hidden-state bean is used, a serialized-object file (.sos) that takes its name from that of the composite class (for example, ivjMyComposite.sos). VisualAge also generates a *getSOSByteArrayCache( )* method, which reads a byte stream from the serialized-object file into an instance variable for deserializing.
- A private method for each method or code connection. For each property-to-property connection where both end points are writable, two methods are generated. Parameter connections appear in code as secondary calls within the original connection.

  The name of the connection method depends on the type of connection you draw. By default, the naming convention is *connFtoFx*, where F represents the type of feature being connected and x is an index number to ensure uniqueness. This translates into the following possible combinations:
  - connEtoM1 (the first event-to-method connection drawn)
  - connEtoS1 (the first event-to-code connection drawn)
  - connPtoP1setTarget (the first property-to-property connection drawn, setting the target from the source)
  - connPtoP1setSource (the first property-to-property connection drawn, setting the source from the target)

  When you rename a connection, VisualAge deletes the obsolete connection method and generates a new method whose name is compatible with the connection's new name.
- For each event set connected to within the composite, listener methods to fulfill the listener interface's implementation requirements. For events in the set that are not currently being used, VisualAge generates stubs only.
- An *initConnections( )* method, which contains a call to every notifier required for event connections in the composite bean. This method is not generated if there are no connections in the composite. For example, if your bean contains a connection from the *actionPerformed* event of Button1, this method contains the call:

  ```
  getButton1().addActionListener(this);
  ```
- For applets, an *init( )* method.
- For nonapplets, an *initialize( )* method.
- A constructor with no arguments, which by default calls the superclass contructor and the *initialize( )* method.

VisualAge also generates the following methods if they do not exist:

- *main(java.lang.String [ ])*. If you copy, rename, or move the bean, you must delete this generated method, regenerate code for the composite, and save the bean.
- For applets, *getAppletInfo( )*.
- *getBuilderData( )*, which contains visual layout information for restoring the free-form surface if the bean is exported and then imported into another VisualAge Java development environment. This method is generated only if the appropriate design-time option has been set in the Workbench.
- *handleException(java.lang.Throwable)*, a stub method for debugging that gets called in *init( )* and the set methods.

Generated Feature Code

The term *feature* refers to an element of the bean interface. Features can be properties, methods, or events. To add features to the interface, use SmartGuides available from the BeanInfo page.

The following items are generated for each feature added from the BeanInfo page.

- For each property, VisualAge generates the following:
  - A declaration for a field, defaulting to package-private access
  - If the property is readable, a get method
  - If the property is writable, a set method
  - If the property is indexed, two additional methods for you to access individual elements
- If at least one property is bound, the *propertyChange* event and associated program elements are generated:
  - *propertyChange*, a field of type *java.beans.PropertyChangeSupport*, declared as protected transient
  - *addPropertyChangeListener(java.beans.PropertyChangeListener)*, a public method
  - *firePropertyChange(java.lang.String, java.lang.Object, java.lang.Object)*, a public method
  - *removePropertyChangeListener(java.beans.PropertyChangeListener)*, a public method
- If at least one property is constrained, the *vetoableChange* event and associated program elements are generated:
  - *vetoableChange*, a field of type *java.beans.VetoableChangeSupport*, declared as protected transient
  - *addVetoableChangeListener(java.beans.VetoableChangeListener)*, a public method
  - *fireVetoableChange(java.lang.String, java.lang.Object, java.lang.Object)*, a public method
  - *removeVetoableChangeListener(java.beans.VetoableChangeListener)*, a public method
- For each event set, VisualAge generates the following items:
  - A declaration for a protected listener field
  - Public addListener and removeListener methods
- For each new listener interface, VisualAge generates the following items. VisualAge gives you the opportunity to specify your own names for these items before they are created.
  - An event class in the same package as the class being edited
  - A listener interface in the same package
  - A multicaster class in the same package
  - An event feature in the class being edited
  - Public addListener and removeListener methods in the class being edited

In addition, VisualAge creates a BeanInfo class when you add the first feature. After that, VisualAge updates the BeanInfo class to reflect each feature changed from the BeanInfo page.

Generated BeanInfo Descriptor Code (an advanced topic)

BeanInfo code defines the public interface of your bean in a standard way. Specifying BeanInfo code enables your bean to be used with any development tool, including VisualAge, that supports the JavaBeans specification. With BeanInfo code available, enabled tools can query an instance of your bean for information about its interface, regardless of underlying implementation.

VisualAge generates BeanInfo class code as needed to capture the interface details you specify on the BeanInfo page for your bean. If a BeanInfo class does not exist for the bean, VisualAge uses the process of *reflection*, matching interface features against Java design templates, to create the class. If a BeanInfo class does exist, VisualAge generates BeanInfo code only for interface elements defined as bean features (properties, methods created from the BeanInfo page, and events). For more information about reflection, see the JavaBeans specification.

The following items are generated when you add features to the bean from the BeanInfo page:

- A *className*BeanInfo class in the same package. This type of class normally contains several list and descriptor methods, which enabled tools call to get information about the bean. A list of the most commonly generated methods follows:
  - *getBeanClass( )*, which returns a instance of *java.lang.Class* that corresponds to the bean.
  - *getBeanClassName( )*, which returns a String whose value is the full name of the bean.
  - *getEventSetDescriptors( )*, which returns an array of descriptors corresponding to the event sets implemented in the bean.
  - *getMethodDescriptors( )*, which returns an array of descriptors corresponding to the method features implemented in the bean.
  - *getPropertyDescriptors( )*, which returns an array of descriptors corresponding to the properties implemented in the bean.
  - *findMethod(Class, String, int)*, used to locate a descriptor method that is requested by name but not found.
  - *getAdditionalBeanInfo( )*, generated only if you opted to inherit BeanInfo from the bean's superclass. (This is set from the Design Time page of the Options notebook.)
- *getBeanDescriptor( )*, which returns general information about the bean:
  - If you marked the bean as expert, this method calls *setExpert(true)*.
  - If you marked the bean as hidden, this method calls *setHidden(true)*.
  - If you require all instances of the bean to be serialized, this method calls *setValue("hidden-state", Boolean.TRUE)*.
- For applets, an appletInfoPropertyDescriptor method used to get information about the *appletInfo* property.
- For each method feature in the bean being browsed, a public *methodName*MethodDescriptor method in the associated BeanInfo class. This descriptor method determines how information about the method is revealed: its true name, its display name, and a description of the bean. If you created the bean in VisualAge, the descriptor method reflects selections you made through the New Method Feature SmartGuide.
- *handleException(java.lang.Throwable)*, a stub method for use in debugging.

- For each property, a public *propertyName*PropertyDescriptor method in the associated BeanInfo class:
  - If you opted to bind the property to an event, this method calls *setBound(true)*.
  - If you marked the property as expert, this method calls *setExpert(true)*.
  - If you marked the property as preferred, this method calls *setValue("preferred", Boolean.TRUE)*.
  - If you marked the property as design-time to keep it from appearing in the property sheet at run time, this method calls *setValue("ivjDesignTimeProperty", Boolean.FALSE)*.
- If at least one property is bound, the following methods:
  - *addPropertyChangeListenerMethodDescriptor( )*, a public method in the associated BeanInfo class
  - *removePropertyChangeListenerMethodDescriptor( )*, a public method in the associated BeanInfo class
- For each method, a public *methodName*MethodDescriptor( ) method in the associated BeanInfo class. If you opted to hide the method, the descriptor method calls *setHidden(true)*.

  Note: If a BeanInfo class already exists, VisualAge does not generate BeanInfo descriptors for methods added from the Methods page.
- For each event set, public event descriptor methods in the associated BeanInfo class (for example, *actionEventSetDescriptor( )* and *actionactionPerformed_javaawteventActionEventMethodEventSetDescriptor( )*)
- For each new listener, a public event descriptor method in the associated BeanInfo class (for example, *stringModifiedEventSetDescriptor ( )*, *stringModifiedSignalModification_CodeGenStringModifiedEventMethodEventDescriptor( )*, and *stringModifiedSignalModification_javalangObjectMethodEventDescriptor( )*).

How Generated Code Coexists with User-Written Code

Generated code falls into the following categories:

> **Items generated only if they do not exist.** These include *main(java.lang.String[])*, *handleException(java.lang.Throwable)*, and *getAppletInfo( )*. You can write your own versions or modify generated versions; VisualAge preserves your code.
>
> **Items regenerated around handwritten changes.** Most code of interest falls into this category. These methods contain the reminder comment `WARNING: THIS METHOD WILL BE REGENERATED.` and are indicated by to the right of each method signature. The rest of this section describes how VisualAge handles this type of generated code.

In general, VisualAge preserves handwritten changes to basic class declarations, as follows:

- *package* and *import* statements associated with the class.
- Access and keyword modifiers for the class.
- Interfaces implemented completely by hand.
- Uniquely named fields, as long as their names do not start with *ivj*. Because VisualAge uses *ivj* to mark generated fields, handwritten fields starting with *ivj* will

be deleted the next time VisualAge generates code for the class. VisualAge does not preserve updates to access modifiers (private, public, protected) in generated fields.

- Uniquely named methods, including exceptions. VisualAge preserves updates to access modifiers (private, public, protected) in generated methods if the updates render access less restrictive.
- Handwritten comments in generated methods.

You can add lines of code in designated areas of generated methods. VisualAge indicates these areas in the generated code with comment lines similar to the following:

```
//user code begin {1}
//user code end
```

If a generated method does not include comment lines like these, any code you add will be overwritten the next time the bean is saved.

RELATED CONCEPTS

“Chapter 2. How Classes and Beans Are Related” on page 3

“Chapter 4. Visual Composition Editor Overview” on page 7

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 19. Connecting Beans” on page 77

“Chapter 27. Defining Bean Interfaces for Visual Composition” on page 131

RELATED REFERENCES

“Chapter 10. Example of Generated Feature Code” on page 39

“Chapter 11. Example of Code Generated from Visual Composite” on page 41

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 10. Example of Generated Feature Code

Suppose you define *simpleString*, the most basic property useful for visual composition: readable, writable, and bound to an event. VisualAge generates the following items:

- *fieldSimpleString*, a field of type *java.lang.String* with default (package-private) access
- *propertyChange*, a field of type *java.beans.PropertyChangeSupport* declared as protected transient
- *getSimpleString( )* and *setSimpleString(java.lang.String)* methods
- *add*- and *removePropertyChangeListener(java.beans.PropertyChangeListener)* methods
- *firePropertyChange(java.lang.String, java.lang.Object, java.lang.Object)*

If *simpleString* is indexed, VisualAge also generates the following items:

- *getSimpleString(int)*
- *setSimpleString(int, java.lang.String)*

Suppose you define an existing event set, *action*, for use in your class using default values in the New Event Set SmartGuide. VisualAge generates the following items:

- *aActionListener*, a field of type *java.awt.event.ActionListener* declared as protected transient
- *add*- and *removeActionListener(java.awt.event.ActionListener)* methods
- *fireActionPerformed(java.awt.event.ActionEvent)*

Suppose you define *stringModified*, a new listener event specifically for your class. VisualAge generates the following items:

- *StringModifiedListener*, an interface.
- *StringModifiedEvent* and *StringModifiedEventMulticaster* classes.
- *add*- and *removeStringModifiedListener(StringModifiedListener)* methods in the class being edited.
- A listener method stub, for example, *signalStringModification(StringModifiedEvent e)*. You must enter a name for this stub yourself.

RELATED CONCEPTS

"Chapter 9. Generated Code" on page 33

"Generated BeanInfo Descriptor Code (an advanced topic)" on page 36

"Code Generated from Visually Composed Beans" on page 33

"How Generated Code Coexists with User-Written Code" on page 37

RELATED TASKS

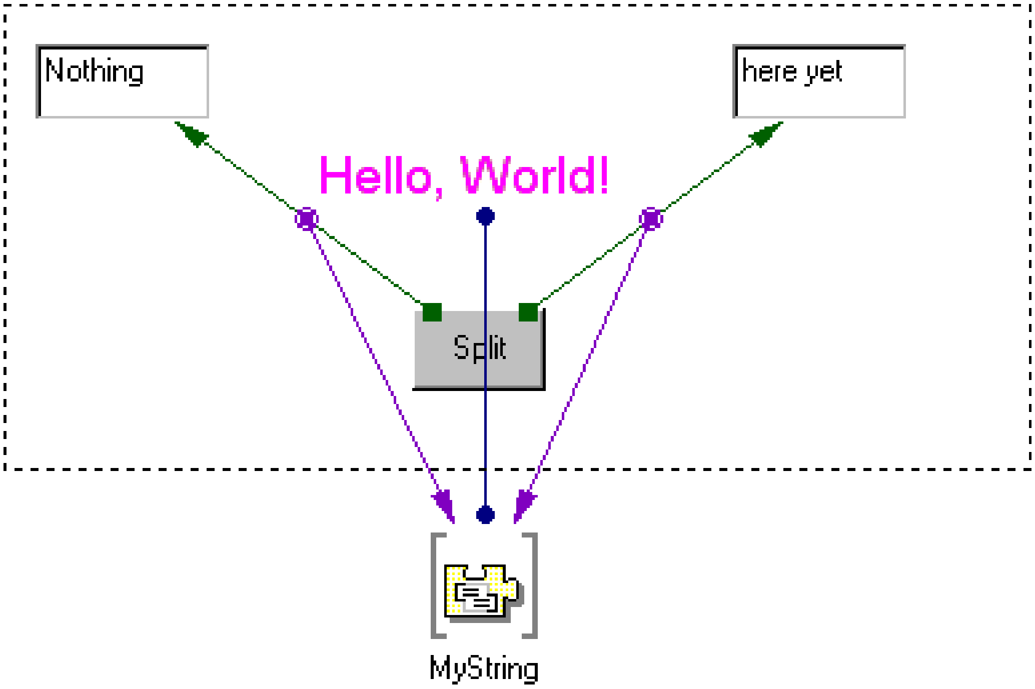
"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

RELATED REFERENCES

"Chapter 11. Example of Code Generated from Visual Composite" on page 41

"Chapter 28. Beans for Visual Composition" on page 141

Chapter 11. Example of Code Generated from Visual Composite



This simple Hello World! applet contains the following beans. The applet subclass itself is represented by the dotted rectangle.

- Two TextField beans, FirstWordArea and SecondWordArea
- A Label bean, MyHello
- A Button bean, SplitButton
- A Variable bean of type *java.lang.String*, MyString

All the finished applet does at run time is split the text value of the Label bean into two substrings and copy each substring to an entry field.

Suppose the composite is saved after the beans have been dropped and edited for initial content. VisualAge generates field declarations and accessor methods as follows:

- *ivjFirstWordArea*; *getFirstWordArea( )*. The initial contents of FirstWordArea (″Nothing″) are set in *getFirstWordArea( )*.
- *ivjSecondWordArea*; *getSecondWordArea( )*. The initial contents of SecondWordArea (″here yet″) are set in *getSecondWordArea( )*.
- *ivjMyHello; getMyHello( )*.
- *ivjSplitButton; getSplitButton( )*.
- *ivjMyString; getMyString( )* and *setMyString(java.lang.String)*.

VisualAge also generates the following applet methods:

- *main(java.lang.String [ ])*
- *init( )*
- *getAppletInfo( )*

- *getBuilderData( )*
- *handleException(Throwable exception)*

The first connection, connPtoP1, links the *text* property of MyHello to the *this* property of MyString. Neither feature is bound. When you save the composite, VisualAge generates these additional methods:

- *connPtoP1SetSource( )*
- *connPtoP1SetTarget( )*
- *initConnections( )*, which calls *connPtoP1SetTarget( )* to initialize the String variable. Because neither property is bound, these methods are not called again unless you connect to them using a code connection.

Connections from the *actionPerformed* event of SplitButton to the *text* property of each TextField bean (connEtoM1 and connEtoM2) reset the text displayed in each entry field. By default, no event data is passed to the target of the connection, so each connection requires input for the new value of *text*. Parameter connections (connEtoM3 and connEtoM4) pass in these values: the *value* property of each push button connection is connected to the *substring(int, int)* method of MyString. The exact character indexes are provided as connection properties of connEtoM3 and connEtoM4.

Now when you save the composite, VisualAge generates these additional methods:

- *connEtoM1(java.awt.event.ActionEvent)*, which calls *getFirstWordArea().setText(getMyString().substring(0, 5))* (changing ″Nothing″ to ″Hello″)
- *connEtoM2(java.awt.event.ActionEvent)*, which calls *getSecondWordArea().setText(getMyString().substring(7,12))* (changing ″here yet″ to ″World″)
- *actionPerformed(java.awt.event.ActionEvent)*, which calls the connEtoM1 and connEtoM2 methods

Suppose you then separate all text into a list bundle. If you opt for a new resource bundle, VisualAge creates a resource class for you in the same package. In addition, the get methods are regenerated to retrieve the appropriate resource instead of using hardcoded text. The call for setting MyHello to ″Hello, World!″ looks like this:

```
ivjMyHello.setText(java.util.ResourceBundle.getBundle
   ("CodeGen.HelloResources").getString("rHelloWorldText"));
```

RELATED CONCEPTS

“Chapter 9. Generated Code” on page 33

“Generated BeanInfo Descriptor Code (an advanced topic)” on page 36

“Code Generated from Visually Composed Beans” on page 33

“How Generated Code Coexists with User-Written Code” on page 37

“Chapter 14. Internationalization in VisualAge” on page 49

RELATED TASKS

Exporting to the file system

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 19. Connecting Beans” on page 77

“Chapter 25. Separating Strings for Translation” on page 127

RELATED REFERENCES

“Chapter 10. Example of Generated Feature Code” on page 39

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 12. Morphing

In the Visual Composition Editor, morphing enables you to change the class or type of a component without significantly reworking property or connection settings. This capability can be very helpful in tasks like the following:

- Changing AWT components to Swing components
- Repairing breakage caused by renaming a class or moving it to a different package
- Changing class beans to Variables (or the reverse)

However, you cannot change the superclass of a composite through morphing.

If renaming or moving a class has introduced a referencing error into a composite, VisualAge alerts you when you attempt to open the composite. If you know that a class has been moved or renamed and you already have the composite open, you can select **Resolve Class References** from the **Bean** menu. In either situation, VisualAge searches the repository for a like-named class and presents the first candidate it finds. You can choose to proceed with the candidate VisualAge suggests, ignore the problem for now, or specify an alternative class.

In situations other than breakage, you change the class or type of a component by selecting **Morph Into** from the bean’s pop-up menu. In this case, you must specify the name of the replacement class or bean type.

Common property settings and connection endpoints are preserved in the new component. To determine feature commonality, VisualAge compares both name and type; both must match. For example, suppose you change a component from *java.awt.TextField* to *com.sun.java.swing.JTextField*. The background color for the original TextField happens to be gray. The gray setting is propagated into the new JTextField.

Conversely, VisualAge discards property settings that cannot be used in the new class. This includes torn-off properties but does NOT include promoted properties. You must delete obsoleted promotions manually from the BeanInfo page of the class browser.

Connection endpoints are handled similarly. Connections to features that are no longer valid in the new class remain until code is regenerated for the composite. To delete such connections instead, select **Delete invalid connections** before you select **OK** to start the morphing process. Because components vary in their use of like-named features, make sure that the updated composite behaves as you expect it to.

RELATED REFERENCES

“Chapter 52. Morph Into” on page 229

“Chapter 53. Resolve Class References” on page 231

Chapter 13. Object Serialization in VisualAge

Object serialization is a means that the Java language provides to save state information about a class instance between runs of a program element.

- Serializable classes support the *java.io.Serializable* interface, which provides a protocol for writing self-contained instance state information to a binary file.
- Externalizable classes support the *java.io.Externalizable* interface, which provides a protocol for writing identity information about an instance to a binary file. This protocol is not complete; it depends on other state information being preserved within the class itself.

To experiment with serialization, start with examples:

- The *com.ibm.ivj.examples.vc.propertyeditors* package of the IBM Java Examples project contains one simple serializable class, called Name. To have VisualAge write a serialization file for you, run the main() method of the NameWriter class.
- The Sun BDK Examples project shipped with VisualAge includes additional examples of both serializable and externalizable classes. The *sunw.demo.buttons* package contains a serializable class (ExplicitButton), an externalizable class (ExternalizableButton), and sample classes (OrangeButtonWriter and BlueButtonWriter, respectively) that can be used to save a sample instance.

Marking a class serializable does not require that it be serialized. It simply gives you the flexibility to do so. For more information about object serialization issues in general, see Sun’s Object Serialization site and the JavaBeans specification.

VisualAge supports two different aspects of serialization:

- **Direct consumption of serialization (.ser) files within the IDE.** Wherever serialized beans are supported, you can enter the name of an .ser file instead of a class name. This includes adding the serialized bean to the beans palette and dropping it into a composite. You can add serialization files to the beans palette in the Visual Composition Editor just as you would bean classes present in your workspace. Once a serialized bean has been dropped, you interact with it as you would any other type of bean.

  Before using a serialized bean, make sure that all classes referenced by the serialized bean exist in your workspace. Otherwise, VisualAge cannot deserialize the bean.
- **Enforcement of required serialization.** You can require that all instances of a serializable bean be saved and restored through serialization by setting its Hidden-state attribute to `true` in its associated BeanInfo class. VisualAge writes all hidden-state beans found within a class to a single serialized-object file (.sos) associated with the class.

VisualAge also uses serialization internally to preserve property settings, so it is important that property values be serializable. By default, primitive types, arrays, strings, and any bean that inherits from *java.awt.Component* are serializable.

RELATED CONCEPTS

“Chapter 9. Generated Code” on page 33

RELATED TASKS

“Chapter 21. Managing the Beans Palette” on page 85

“Creating and Modifying a BeanInfo Class” on page 132

RELATED REFERENCES

“Chapter 43. Choose Bean Window” on page 211

Chapter 14. Internationalization in VisualAge

VisualAge 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. *ListResourceBundle.getContents( )* returns an array of key-value pairs. The key-value pairs stored as static strings in a property file are used to initialize the corresponding bean when it is loaded. 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 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. You can edit existing resource bundles by hand or from within VisualAge. Multiple resource sources can be referenced within a single bean. VisualAge generates the appropriate code the next time you save the bean.

VisualAge does not separate text located in user code fields. To take advantage of programmatic string separation, move the user code into a separate method and call the method from within the user code block.

RELATED TASKS

"Chapter 25. Separating Strings for Translation" on page 127

RELATED REFERENCES

java.util package

Chapter 15. Using Visual Composites Imported from Version 1

If you imported composites using a Version 1 interchange file (.dat), you must open them in the Visual Composition Editor and resave them in the current environment before exporting again. This is necessary because the scheme through which VisualAge records visual composition information changed after Version 1.

In Version 1, you shared composites with other repositories by exporting to an interchange file distinct from Java source. Starting in Version 2, you share composites by exporting to a Java source file. Visual composition information now resides in a method within the class, named *getBuilderData( )*. This method is regenerated every time the composite is saved. Your resave operation gives VisualAge the opportunity to generate this method the very first time.

- RELATED CONCEPTS
- “Code Generated from Visually Composed Beans” on page 33

Chapter 16. Working with Beans Visually

You can perform the following tasks to modify your beans:

- “Editing Beans within a Composite Bean”
- “Renaming Beans and Connections”
- “Working in the Beans List” on page 65
- “Setting the Tabbing Order” on page 54
- “Promoting Bean Features” on page 55
- “Tearing Off Properties” on page 56
- “Editing Bean Properties” on page 56
- “Setting a Layout Manager during Visual Composition” on page 60
- “Editing Bean Labels” on page 60

Editing Beans within a Composite Bean

VisualAge enables you to edit a composite bean or nonvisual bean that is embedded within another composite bean.

To modify the bean, open the bean pop-up menu and select **Open**. The Visual Composition Editor appears for that bean. If you add features to the embedded bean from the BeanInfo page, select **Refresh Interface** when you return to the original Visual Composition Editor.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

RELATED TASKS

“Chapter 17. Composing Beans Visually” on page 67

“Chapter 18. Manipulating Beans Visually” on page 71

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Renaming Beans and Connections

The Visual Composition Editor assigns default names to distinguish beans and connections when you generate the code to build your program element. For example, for each bean, VisualAge generates a get method that you should use to access the bean within user methods or code. In addition to use in generated methods for beans and connections, the names for selected beans and connections appear in the information area at the bottom of the Visual Composition Editor.

The Visual Composition Editor assigns bean names based on the bean palette name or the name you specify when you use the Choose Bean tool. For example, VisualAge names the first button bean that you drop `Button1`, the second `Button2`, the third `Button3`, and so forth. When you select this bean, the information area at the bottom of the Visual Composition Editor displays the message `Button1 selected`.

The Visual Composition Editor assigns connection names based on the type of connection. For example, the first property-to-property connection is named *connPtoP1*, the second is *connPtoP2*, etc. All other connection types receive their names simlarly. For example, an event-to-code connection is named *connEtoC1*, an event-to-method is named *connEtoM1*, etc.

To assign bean or connection names that are more descriptive or meaningful to your program element, follow these steps:

1. Move the mouse pointer over the bean or connection that you want to rename.
2. Click mouse button 2 and the pop-up menu appears.
3. Select **Change Bean Name** or **Change Connection Name** and the Name Change Request or Connection Name Change Request window appears.
4. Type a new name in the entry field.
5. Click **OK**. The Visual Composition Editor changes the name.

To view the changes outside the Visual Composition Editor, you must regenerate the code.

You can also rename beans in the property sheet and you can change bean or connection names from the pop-up in the Beans List.

Note: When you name a bean, take into account that VisualAge uses the name of the bean in generating get methods. For example, if you name a torn-off property *font*, the generated method *getFont* overrides the inherited method.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

RELATED TASKS

“Opening the Property Sheet for a Bean” on page 56

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Setting the Tabbing Order

The tabbing order specifies how the input focus moves from bean to bean as the user presses the Tab key.

1. Open the pop-up menu for the composite bean.
2. Select **Set tabbing** and then **Show tab tags** and numbered tab tags appear.
3. Place the mouse pointer over the tab tag you want to change.
4. Press and hold mouse button 1.
5. Drag the tab tag to its new position.
6. Release mouse button 1.
7. Repeat until the numbered tags reflect the tabbing order you desire.

You can also change the tabbing order by moving the beans in the Beans List.

RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

“Setting Tabbing Order” on page 12

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Promoting Bean Features

If you want to connect to the features of a bean that is embedded within a composite, you must promote bean features. To promote the bean features:

1. Open the bean pop-up.
2. Select **Promote bean feature**.
3. Select the type of feature: **Method**, **Property**, or **Event**.
4. From the list box, select the feature you want to promote and **>>** is enabled.
5. Click **>>** and the feature is listed in the **Promoted features** list.
6. Repeat for the other features you wish to promote.
7. If you wish to rename the feature, double-click and modify the name listed in the **Promote Name** list.
8. If you wish to remove the feature from the **Promoted features** list, click **<<**.
9. Click **OK**.

When you promote features, VisualAge performs the following tasks:

- Saves the composite bean
- Generates the code for the composite bean
- Creates a BeanInfo Class, if one doesn’t already exist
- Adds a new connection between the embedded bean and the composite, where appropriate (for all but promoted methods)

When you embed this bean within another bean, the features that you promoted are listed under the type in the **Connectable Features** window. Connect the promoted feature of each bean to effect the change you desire.

RELATED CONCEPTS

“Promotion of Bean Features” on page 23

“Chapter 4. Visual Composition Editor Overview” on page 7

RELATED REFERENCES

“Promote Bean Feature” on page 205

“Chapter 44. Promote Features Window” on page 213

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

Tearing Off Properties

If you want to gain access to an encapsulated feature of a bean, you must tear off the property. To tear off a property:

1. Select **Tear off property** from the pop-up menu of the bean with the property you want to access. Another menu appears listing all of the properties of the bean.
2. Select the property you want to tear off. The mouse pointer is now loaded with a variable bean representing or pointing to the property you selected.
3. Place the new bean on the free-form surface, as you would any other nonvisual bean. The torn-off property now appears as a variable bean connected to the original bean by a property-to-property connection.

RELATED CONCEPTS

"Tearing Off Properties" on page 13

"Chapter 4. Visual Composition Editor Overview" on page 7

RELATED REFERENCES

"Chapter 38. Visual Composition Editor" on page 191

"Chapter 41. Pop-Up Menus for the Visual Composition Editor" on page 201

Editing Bean Properties

The property sheet for a bean provides a way to display and set initial property values for the bean. Changes made to values in the property sheet are applied immediately.

You can edit the properties for a single bean or select several beans and open a property sheet for them. When you change a property on the property sheet, the change affects all the selected beans.

Some of the tasks you can perform include:

- "Using Code Strings in Bean Properties" on page 57
- "Changing Bean Colors" on page 58
- "Changing Bean Fonts" on page 58
- "Changing Bean Size and Position" on page 59
- "Adding Bean Icons" on page 59
- "Setting a Layout Manager during Visual Composition" on page 60
- "Editing Bean Labels" on page 60

Opening the Property Sheet for a Bean

To change the properties for a single bean, follow these steps:

1. From either the free-form surface or the Beans List window, place the pointer over the bean and double-click mouse button 1. The property sheet for the bean appears.
2. When you have chosen the property you want to modify, select the value field to the right of the property name.

3. Make the appropriate changes from the provided options. Options for modifying the properties depend upon the bean and may include selecting a button, entering information into the field, selecting from a drop-down list, or proceeding to other dialog boxes.

To change the properties for multiple beans (multiple selection), follow these steps:

1. Select the beans with properties you want to change.
2. Move the mouse pointer over one of the selected beans.
3. Click mouse button 2.
4. Select **Properties** from the pop-up menu. A property sheet for the selected beans appears and displays the common properties for the selected beans.

Once you open a property sheet, you can modify properties for most beans in the Visual Composition Editor. To open the properties on another bean select it in the Visual Composition Editor. To select if from within the property sheet:

1. Open the drop-down list at the top of the property sheet.
2. Select the bean you want to modify.
3. Modify the properties.

Note:

> If a common property is not visible on the property sheet, select the **Show expert features** check box.
>
> To enable national language support for your composite, be sure to read “Chapter 25. Separating Strings for Translation” on page 127.

Using Property Interface Editors

Some bean properties on the property sheet, such as border and model, use interface editors. If the property has an interface editor, a small button, ... which indicates a secondary window, appears to the right when you select the value field for that property. When you select the ... , the interface editor for that property appears.

The interface editor provides two options: **Code String** and **Bean Implementing Interface**. To enter your own code string for the property, select **Code String** and type your code in the entry field. To see a list of available interfaces, select **Bean Implementing Interface**. From the drop-down list, select the desired interface for that property and modify any values listed.

Using Code Strings in Bean Properties

VisualAge enables you to set some bean properties using code strings. Curly braces, **{ }**, in the value column of a bean property sheet indicate that you can type Java code directly into the property field.

For example, you can use code strings to dynamically instantiate a dialog window when the user selects a button.

1. Create and save a dialog window named *MyDialog*.

2. Create the main user interface bean and drop a factory bean with the type *MyDialog*.
3. Connect the *actionPerformed* event of a button to the *MyDialog* method of the factory.
4. Open the Event-to-Method Connection window and select the **Set parameters** button.
5. Enter *new java.awt.Frame()* in the value field of the parent property.

   This string directs VisualAge to create a new instance of *MyDialog* when the button is selected.
6. Connect the *actionPerformed* event of the button to the *Show()* method of the factory, to make the dialog visible.

Changing Bean Colors

Visual beans have color properties for the foreground and background. To change a color property of a bean in its property sheet, do the following:

1. Select either the *foreground* or *background* property. A small selection button ... appears in the value column.
2. Click the ... button to open the Colors property window.
3. Select either the **Basic**, **System**, or **RGB** checkbox.
   - **Basic**—Select either a color box or the color name. The selected color appears in the preview window.
   - **System**—Select the standard system object color.
   - **RGB**—Manipulate the sliders to create the desired color.
   - Click **OK** to accept the changes and return to the property sheet.

When you change bean colors, select colors that are available across various platforms.

For an example, open the background property in the property sheet for any of the *com.ibm.ivj.examples* shipped in the IBM Java Examples project.

Note: If you are setting the background property of a Swing bean, be sure the opaque property of the bean is set to true.

Changing Bean Fonts

To change the font of a bean in its property sheet, do the following:

1. Select the *font* property. A small selection button ... appears in the value column.
2. Click the ... button and the Fonts property window appears.
3. Modify any of the following font values:
   - Name
   - Style
   - Size
4. Preview the font changes in the preview window.

5. Click **OK** to apply the changes and return to the property sheet.

When you change bean fonts, select fonts that are available across various platforms.

For an example, open the font property in the property sheet for any *com.ibm.ivj.examples.vc.* class shipped in the IBM Java Examples project.

Changing Bean Size and Position

If a bean is embedded in a container that does not use a layout manager (null layout), you can change the x and y coordinates, the width, and the height using the bean property sheet. To modify the size and position features, do the following:

1. Click the expansion icon ⊞ to the left of the *constraints* property.
2. Select the value field for the property you want to modify and enter the new property value.
3. Press enter to apply the value.

Note: If you specify a non-null layout for the container, the bean position and sizing constraints are affected by that layout manager.

When you open the bean property sheet using multiple selection, some size and position constraints may appear stippled rather than solid because the values for the field are not common to all the selected beans. Once you modify the constraint, however, all the selected beans have the same value and the constraint appears solid.

For an example of the different layout managers, see the *COM.ibm.ivj.examples.vc.layoutmanagers.LayoutManagers* class shipped in the IBM Java Examples project.

Adding Bean Icons

Some visual beans, such as JLabel and JButton, have icon properties. You can assign an icon to these beans through the property sheet.

To add an icon to a visual bean:

1. Open the bean property sheet.
2. Select the Icon value field.
3. Select ... and the Icon Editor appears.
4. To add an icon from a graphics file, select **File** and enter the fully qualified path and file name in the name field, or select **Browse** and select the file through the file locator.
5. To add an icon from a URL, select URL and enter the fully qualified path and file name in the name field.
6. Press Enter to view the icon in the preview pane.
7. Select **OK** to accept the icon.

Editing Bean Labels

Some visual beans, such as buttons and menus, contain text strings. You can edit these labels through the property sheet.

To edit the text of a label:

1. Open the bean property sheet.
2. Select the label value field.
3. Enter the new label name.

RELATED CONCEPTS

"Chapter 4. Visual Composition Editor Overview" on page 7

"Property Sheets" on page 11

RELATED TASKS

"Chapter 18. Manipulating Beans Visually" on page 71

RELATED REFERENCES

"Chapter 38. Visual Composition Editor" on page 191

"Chapter 41. Pop-Up Menus for the Visual Composition Editor" on page 201

Setting a Layout Manager during Visual Composition

To set a layout manager in a container bean, follow these steps:

1. Open the property sheet for the container bean.
2. From the value column, select the cell for the *layout* property.
3. From the **Layout Manager** drop-down list, select a layout.

Setting Layout Properties during Visual Composition

The complexity of this task depends on which layout manager you choose for your container bean and how much custom behavior your bean requires. Default behavior exists to some extent for all layouts. To set layout properties for the components of the container bean, follow these steps:

1. Open the property sheet for the component bean.
2. If one exists, click the expansion icon ⊞ to the left of the *constraints* property.
3. Select the value field for the property you want to modify and enter the new property value.
4. Press enter to apply the value.

Consider waiting to set layout properties until you have settled on a layout manager. Many values are lost when you switch layouts or move the component to another container.

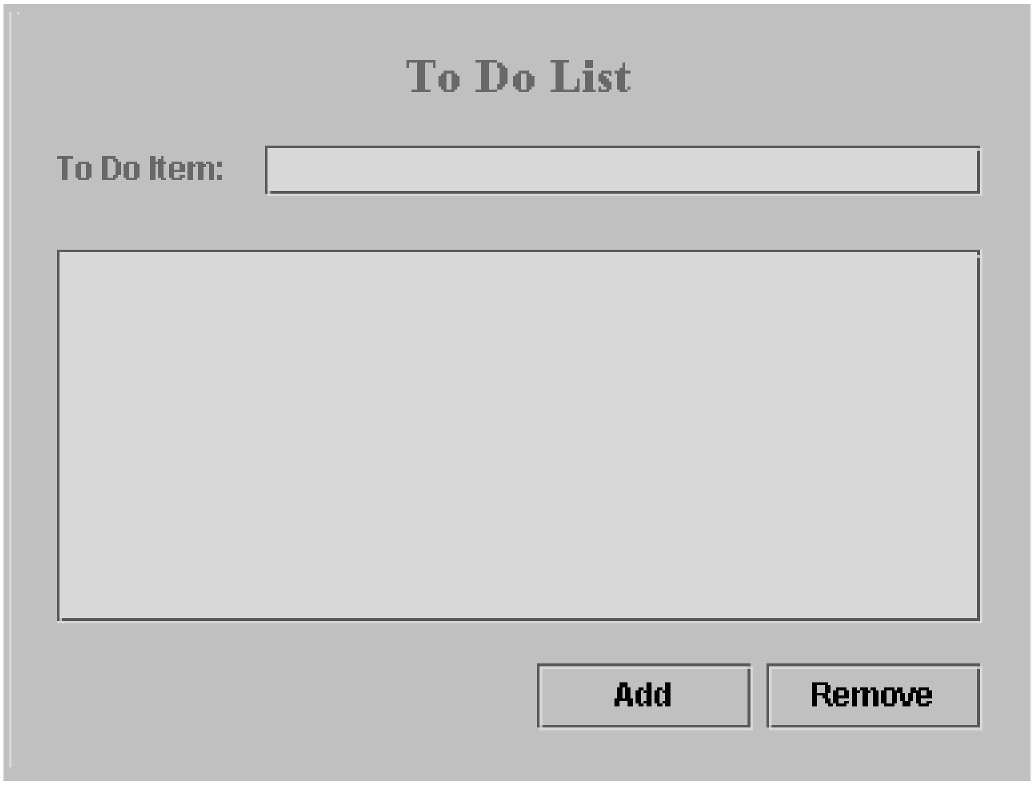
GridBag Layout Constraints

GridBag is a powerful tool for layout, however, because of its many choices, it can become confusing. Once you have selected GridBag layout for your user interface, adjust the constraints to affect the placement of beans within the layout.

Constraint	Specifies...
Anchor	Where within the cell (North, South, etc.) to place a component that is smaller than the cell.
Fill	How to size the component (Vertical, Horizontal, Both, or None) when the cell is larger than the component's requested size.
gridX, gridY	The column (gridX) and row (gridY) where your component resides. If gridWidth or gridHeight is >1, gridX and gridY are the starting row and column. The default places a new component to the right or below the previously added component.
gridWidth, gridHeight	The number of cells the component should use in a row (gridWidth) or column (gridHeight).
insets	The component's external padding, the minimum amount of space between the component and the edges of its cell.
ipadX, ipadY	The component's internal padding within the layout, or how much to add to the width and/or height of the component.
weightX, weightY	How to redistribute space for resizing behavior. You should specify a weight for at least one component in a row (weightX) and column (weightY) to prevent all the components from clumping together in the center of their container. The distribution of extra space is calculated as a proportional fraction of the total weightX in a row and weightY in a column.

Creating a GUI Using GridBagLayout

Follow these steps to create a To Do List GUI using GridBagLayout:



1. To set up the applet:
   - From your package in the IDE add an applet with JApplet as the superclass.
   - Open the Visual Composition Editor to the JApplet. The applet contains a JAppletContentPane, on which you work.
   - Select from the tool bar, and place the Beans List window in a visible but out of the way location.
   - Select from the tool bar, and place the Properties window in a visible but out of the way location.

     Note: These windows are necessary for modifying properties and accessing covered components throughout this process.
   - Select the JApplet.
   - From the Properties window, open the layout drop-down, and select GridBagLayout.
2. To place beans in the GridBag:
   - Select a JLabel from the palette and drop it on the JAppletContentPane. Because GridBagLayout distributes extra space around the beans in the layout, the JLabel falls to the center of the pane. You change this behavior, later in the procedure, by setting the weightX/weightY constraints property on one of the components in the layout.
   - Select a JTextField.
   - Hold mouse button 1 down and move the pointer to the right of the JLabel. The dark line between the pointer and the JLabel indicates target emphasis.

Release the mouse button while the emphasis line is on the right side of the JLabel and the JTextField appears to the right of the JLabel.

Note: Beans dropped in GridBagLayout appear in their *preferred size*. The preferred size, which is different for every component, is usually dependent on one of the component's property settings. For example, the preferred size of JLabel is the size of the text within the label. If the component is a container, like a JPanel or a JScrollPane, the preffered size usually reflects the preffered size of the contents of the component.

- Select another JLabel and position it so that target emphasis is on top of the JLabel1.
- Select a JScrollPane and position it so that target emphasis is under the JLabel1.
- Select a JList and place it on the the JScrollPane in the beans list. Because the preferred size of the JScrollPane is small, it is easier to add the JList bean directly onto the JScrollPane that is in the Beans List window. When you drop the bean, the preferred sizes of the JScrollPane and JList align.

3. To adjust property values within the Properties window, select the bean and then adjust the property value:

Bean...	**Property...**	**Change Value to...**	**What You See..**
JTextField1	columns	5	TextField width increase
JLabel1	text	To Do Item:	Change in text
JLabel2	text	To Do List	Change in text
JLabel2	font	Name—serif, Size—18	Change in font style and size
JLabel2	horizontalAlignment	center	Label placement is centered

4. To adjust the constraints properties within the Properties window, select the bean and then open the constraints tree-view and adjust the property value :

Note: Since the JScrollPane, not the JList, is the component that is sitting in the GridBagLayout, you modify its constraints. Because the JList completely covers the JScrollPane, you must select the pane from either the Beans List window or the Properties window. If you select the JList instead, you cannot adjust the component width within the layout manager.

Bean...	Constraint...	Change Value To...	What You See..
JScrollPane1	gridWidth	2	Pane covers two cells width
JScrollPane1	fill	both	Pane fills the cell vertically and horizontally
JScrollPane1	weightX	1	Column containing the JScrollpane expands to fill the extra horizontal space in the JApplet

JScrollPane1	weightY	1	Row containing the JScrollpane expands to fill the extra vertical space in the JApplet
JScrollPane1	insets	left—15, bottom— 5, right—15	Padding on the left, bottom, and right of the pane
JLabel2	gridWidth	2	Label covers two cells width
JLabel2	insets	top—15	Padding on the top of the label
JTextField1	fill	horizontal	Textfield fills cell horizontally
JTextField1	insets	top—15, bottom—5, right—15	Padding around top, bottom and right side of the TextField
JLabel1	ipadX	10	Internal padding between text and component border
JLabel1	insets	top—15, left—15, bottom—5, right—5	Padding around all sides of label

Note: Setting the weightX and weightY on the JScrollPane, insures that the GUI resizes proportionately. Setting the ipadX on JLabel1, insures that the label resizes proportionately. This is especially helpful when translating your GUI into other languages.

5. To add the button panel:
   - Select a JPanel and position it so that target emphasis is under the JScrollPane1.
   - Because JPanel defaults to null layout, it has no constraints and is not visible. From the Beans List window, select JPanel1 and change the layout to GridLayout in the Properties window.
   - Place two JButtons on JPanel1 in the Beans List window.
6. To adjust the properties and constraints properties for JPanel1 and its contents:

Bean...	Property...	Change Value To...	What You See...
JPanel1	constraints—anchor	EAST	Panel placement moves to the lower right corner
JPanel1	constraints—insets	bottom—15, right—15	Padding around the right and bottom of the panel
JPanel1	layout—hgap	5	Space between the buttons
JButton1	text	Add	Button label changes
JButton2	text	Remove	Button label changes

RELATED CONCEPTS

“Layout Managers in Visual Composition” on page 14

RELATED TASKS

“Editing Bean Properties” on page 56

RELATED REFERENCES

“Properties” on page 196

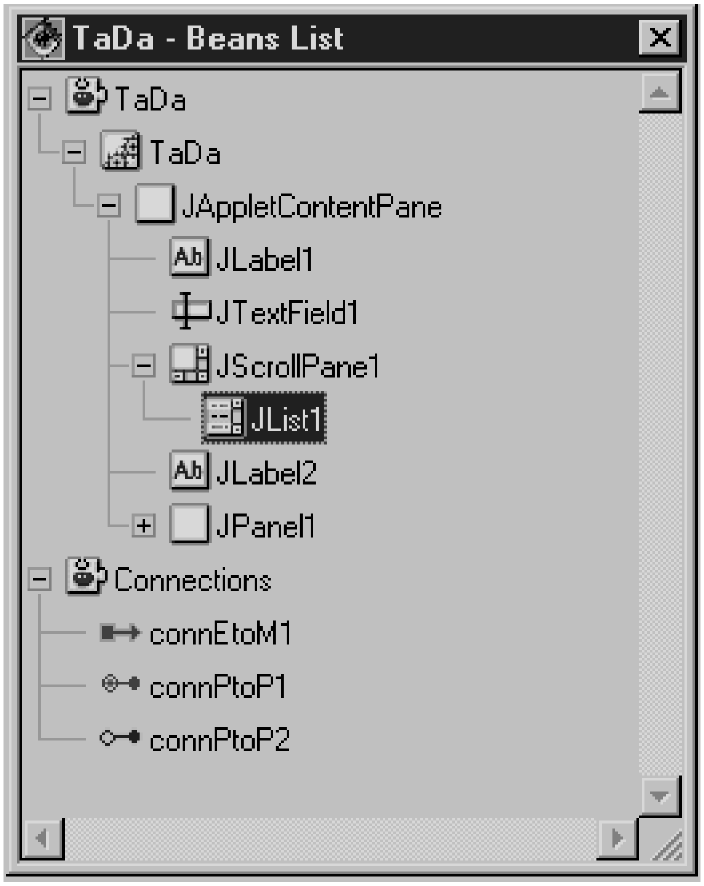
Working in the Beans List

The Beans List window displays an ordered list of the beans and connections on the free-form surface. The beans are initially listed in the order in which they were dropped, which also reflects the tabbing order. If you change the order of beans that have tabbing set, the Visual Composition Editor reflects the updated tabbing order.

To view the beans list, select **Beans List** from the tool bar or select from the Tools pull-down menu.

You can perform the following tasks within the beans list:

- Drop a bean from the palette directly onto the beans list.
- Select a bean or connection.
- Reorder a bean or connection by dragging.
- Change the tabbing order.
- Move a bean to a different composite by dragging.
- Make connections between beans on the beans list.
- Perform any tasks on the bean or connection pop-up for an item in the beans list.



RELATED CONCEPTS

“Chapter 4. Visual Composition Editor Overview” on page 7

RELATED TASKS

“Chapter 18. Manipulating Beans Visually” on page 71

RELATED REFERENCES

“Chapter 38. Visual Composition Editor” on page 191

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

“Beans List” on page 196

Chapter 17. Composing Beans Visually

Visually composing beans means using the Visual Composition Editor to place and connect individual and composite beans in a graphical user interface (GUI). More specifically, visually developing a user interface bean includes:

1. Designing the user interface.
2. Choosing a layout for the bean.
3. Dropping visual and nonvisual beans on the free-form surface or the beans list.
4. Changing properties and manipulating the beans.
5. Making connections to determine the behavior of the beans.
6. Manipulating connections.
7. Running, which includes saving, generating code, and compiling the class.
8. Making changes.

When you edit the bean that represents the overall structure of your application (usually its main user interface view), you graphically build your application. By making connections between beans, you build your program’s business logic.

Embedding Beans in a Composite Bean

A composite bean is a bean that contains other beans. The beans you add are referred to as embedded beans. You can embed primitive or composite beans into your composite bean.

You can add beans that appear on the beans palette, as well as beans that do not appear on the palette. Beans that do not appear on the palette may include composite beans you created, such as a panel with several buttons.

Note: If you use a layout that allows a bean to completely cover another bean, the “Beans List” on page 196 enables you to easily perform tasks on the covered components.

Adding Beans From the Palette

To add a bean from the palette to the Visual Composition Editor surface, follow these steps:

1. From the category menu button, select the category containing the bean you want.
2. Select the bean you want. The mouse pointer becomes a crosshair, indicating that the mouse pointer is loaded with the bean you selected.
3. Move the crosshair to the location where you want to place the bean.
4. Press and hold mouse button 1. An outline of the selected bean appears under the crosshair. Without releasing the mouse button, move the crosshair to position it precisely.
5. Release the mouse button. The bean you selected is placed at the location of the crosshair, and the mouse pointer returns to normal.

   Note: If you specify a non-null layout for the container, the bean placement is affected by that layout manager.

To add a bean from the palette to the beans list, follow these steps:

1. Select the category containing the bean you want.
2. Select the bean you want.
3. From the Beans List window, click on the composite bean that you want to place the bean within. The bean you selected is added to the beans list and the composite bean.

Note: The layout you specified for the container affects the placement for the bean. To modify bean placement on the Visual Composition Editor from within the Beans List window, open the Properties for the bean and modify the **Constraints**.

To add multiple instances of the same bean, enable **Sticky** by holding Ctrl while selecting the bean. Selecting a new bean or the Selection tool disables **Sticky**.

If the bean you want to add is not on the beans palette, you can add it with the Choose Bean tool from the beans palette.

Adding Beans Not on the Palette

You can add a bean as a *class*, a *serialized bean*, or a *variable*. When you add a bean as a *class*, the default constructor for the class is used when the program runs. This means that a real object is created, not a variable that points to a real object defined elsewhere. For more about serialized beans, see “Chapter 13. Object Serialization in VisualAge” on page 47.

Before you try to add a serialized bean, make sure its serialization file (.ser) is somewhere in the classpath for your workspace.

To place a bean on the Visual Composition Editor:

1. From the beans palette, select the **Choose Bean** tool; the Choose Bean window appears.
2. Enter the fully qualified class name in the **Class name** field. The **Browse** button is especially helpful in locating the **Class name** when several of the same name exist in multiple packages.
3. Type a name for the bean in the **Name** field. This name appears in the information area at the bottom of the Visual Composition Editor when you select the placed bean; it represents the bean in generated bean code.

   The **Name** field is optional. If you leave it blank, VisualAge generates a default name based on the class.
4. To close the Choose Bean window after loading the mouse pointer with the bean, click **OK**.

   To enable the **OK** push button, you must enter the fully qualified name, package, and class in the **Class name** field.
5. Move the crosshair to the desired location on either the Visual Composition Editor surface or the beans list, and click mouse button 1.

If you are dropping a bean that uses a graphic resource, place the graphic resource file in the directory where your program element is located. For example, if your program element is called *MyProject.MyPackage.MyApp*, place your graphic resource file in **x:\..\ide\project_resources\MyProject\MyPackage\MyApp** (where x is the drive where VisualAge is installed). If you have not exported your program element or created subdirectories in the project_resources directory, you may need to create the subdirectories manually.

Unloading the Mouse Pointer

To unload the mouse pointer at any time, from the beans palette, click the Selection tool.

Undoing and Redoing Changes in the Visual Composition Editor

If you undo an operation that you decide you had right in the first place, select **Redo** from the **Edit** pull-down menu. **Redo** restores the bean to the state before the last **Undo**, including any connections that were deleted.

Undo and **Redo** affect operations you perform on the free-form surface and beans palette in the Visual Composition Editor. They do not affect any of the functions in the **File** pull-down menu.

Use **Undo** to reverse any or all of the changes you made to the Reorder Connections list.

Saving a Bean

Saving a bean that you have constructed includes generating the source code. To save the bean and generate the source code:

1. Select **Bean** from the menu bar.
2. Select **Save Bean**.

Note: Clicking the **Run** button from the tool bar also saves the bean and generates the source code.

Running and Testing Beans

When you select **Run** from the tool bar, VisualAge performs the following actions on your visually composed bean:

- Saves the bean
- Generates code
- Compiles the class
- Runs the compiled bean in an applet window

RELATED CONCEPTS

"Adding Beans in the Visual Composition Editor" on page 10

"Beans Palette" on page 9

"Free-Form Surface" on page 8

"Layout Managers in Visual Composition" on page 14

RELATED TASKS

“Chapter 21. Managing the Beans Palette” on page 85

“Moving Beans” on page 74

“Editing Bean Properties” on page 56

“Composing an Applet” on page 89

“Changing Bean Size and Position” on page 59

RELATED REFERENCES

“Beans List” on page 196

“Chapter 43. Choose Bean Window” on page 211

Chapter 18. Manipulating Beans Visually

Once you have placed beans on the free-form surface, you can make the following changes to achieve the look and function you want:

- “Selecting and Deselecting Beans”
- “Positioning Beans” on page 72
- “Resizing Visual Beans” on page 73
- “Moving Beans” on page 74
- “Copying Beans” on page 74
- “Deleting Beans” on page 75
- “Displaying Bean Pop-Up Menus” on page 75

Selecting and Deselecting Beans

To select a single bean, click the bean with mouse button 1. If you previously selected other beans, they are deselected automatically.

To select multiple beans, do one of the following:

- For Windows platforms: Hold down Ctrl or Shift and click mouse button 1 on each additional bean you want to select.
- For Windows platforms: To select all the beans within a container, select the container bean and then, from the File pull-down, select Select All.
- For UNIX platforms and OS/2: Hold down mouse button 1 and move the mouse pointer over each bean you want to select. After you select the beans, release mouse button 1.

You can select beans or connections, but not both together. However, if you delete a bean with connections, the Visual Composition Editor deletes the connections and the bean.

When you select a bean in the Visual Composition Editor, selection handles ■ appear on the corners and between the corner handls. If you select more than one bean, the last bean selected has solid selection handles, indicating that it is the *anchor* bean and the other selected beans have hollow selection handles. The anchor bean is the guide by which the other beans are manipulated. For example, if you want to match the widths of two beans, the anchor bean is used as the guideline width.

To change the anchor bean, hold Shift and click the already selected new anchor bean.

Note: You can also select and modify beans, one at a time, in the beans list.

To deselect a bean after you have selected it, click mouse button 1 anywhere on the Visual Composition Editor.

To deselect multiple but not all selected beans, follow these steps:

1. Hold down the Ctrl key.

2. Click and release mouse button 1 on all the beans you want to deselect.

You can deselect all selected beans by clicking mouse button 1 anywhere but on a selected bean.

Positioning Beans

Positioning a bean refers to aligning or spacing. For beans in a null layout, the tool bar and the **Tools** pull-down menu provide options for aligning beans.

The anchor bean, indicated by solid selection handles, is the bean that serves as the alignment reference. To align beans with one another, select the ones you want to modify and select the anchor bean last. You can also change the anchor bean by holding Shift and clicking on the new anchor bean.

To align beans:

1. Select all the beans you want to align, and then select the bean you want the others to match.
2. Select one of the following alignment tools from the tool bar:

   Align Left

   Align Top

   Align Center

   Align Middle

   Align Right

   Align Bottom

The pop-up menu provides options for spacing within the bounding box, an unseen box that contains the selected beans. You can also manage the placement of parts by using a layout managerh.

To space beans within the bounding box:

1. Select all the beans you want to evenly space. You must select a minimum of three beans.
2. From the pop-up menu of one of the selected beans, select **Layout** and then **Distribute**, and then either **Horizontally in bounding box** or **Vertically in bounding box**.

Note: If you specify a non-null layout for the container, the bean alignment and spacing are controlled by that layout manager and not the alignment tools.

Resizing Visual Beans

You can change the size of a visual bean in the Visual Composition Editor using any of the following techniques:

- Dragging the selection handles
- Matching by multiple selection
- Changing Constraints properties
- In a non-null layout manager, resetting layout constraints

Beans that cannot be resized, such as variables, menus, and tear-off properties have reversed background color, but no selection handles. Beans using non-null layouts cannot be resized, but have selection handles.

Note: If you specify a non-null layout for the container, the bean sizing is affected by that layout manager.

Resizing Beans by Dragging

To change the size of a visual bean in a container using a null layout, follow these steps:

1. Select the bean by clicking it with mouse button 1. To size several beans at once, select all the beans you want to size.
2. Place the mouse pointer over one of the handles and hold down mouse button 1.
3. While holding down mouse button 1, drag the handle to a new location. As you move the mouse, the outline of the bean dynamically changes size. When it is the size you want, release the mouse button. The bean changes to the size that you chose.

Note: Pressing the Esc key before releasing the mouse button cancels resizing without making changes.

To size a bean in only one direction, press and hold the Shift key while sizing the bean. Holding down the Shift key prevents one dimension of the bean from changing when you resize the other dimension. For example, to change the width of a bean but prevent its height from changing, hold down the Shift key while changing the width.

Matching Bean Sizes Using the Tool Bar

1. Select all the beans you want to size, making sure the last bean you select, the anchor, is the size you want the others to match. You can change the anchor by holding Shift and clicking the new anchor.
2. Select one of the following from the tool bar or the **Tools** pull-down menu:

   Match Width

   Match Height

   The size of the selected beans changes to match the size of the anchor bean.

Moving Beans

To move a bean in the Visual Composition Editor, follow these steps:

1. Place the mouse pointer over the bean you want to move.
2. Hold down the appropriate mouse button and move the mouse pointer to the new location.
   - In Windows, hold down mouse button 1.
   - In OS/2, hold down mouse button 2.
   - In UNIX platforms, hold down the middle mouse button.
3. Release the mouse button. The bean appears in its new location with a solid border around it, indicating it is selected.

If the bean you are dragging is one of several that you selected, all selected beans move together. Pressing the Esc key before releasing the mouse button cancels the move without making changes.

You can reorder or reparent a bean in the beans list. To reorder, select a bean in the beans list and drag it to a new position within its composite bean. This action does not change the position of the bean in the Visual Composition Editor (except as noted below), but reorders the list, which affects the tabbing order. To reparent a bean, select and drag it to a different composite bean in the beans list. This action does change the position of the bean in the Visual Composition Editor. You cannot, however, select multiple beans on the beans list.

To modify bean placement on the Visual Composition Editor from within the Beans List window without reparenting, open the Properties for the bean and modify the Constraints.

Note: If you use BoxLayout, FlowLayout, or GridLayout managers, you can move beans on the Visual Composition Editor surface by reordering them in the beans list.

Moving a composite bean requires special handling. For information, see “Code Generated from Visually Composed Beans” on page 33.

Copying Beans

To copy beans using the clipboard:

1. Select all the beans you want to copy.
2. From the **Edit** pull-down menu, select **Copy**. A copy of each selected bean is placed on the clipboard.
3. From the **Edit** pull-down menu, select **Paste**. The mouse pointer turns into a crosshair, indicating that it is loaded with the copied beans.

   If you decide against pasting the beans, unload the mouse pointer by selecting the ↖ Selection tool.
4. Position the mouse pointer where you want the beans to be copied.
5. Click mouse button 1. Copies of the beans appear at the position of the crosshair.

As long as you do not copy another item to the clipboard, you can continue copying these beans.

Note: When you copy or cut and paste beans, the Visual Composition Editor preserves the bean names but not the connections.

When you copy or cut and paste two or more beans, they retain their positions relative to each other.

To copy beans by dragging:

1. Select all the beans you want to copy. If you only want to copy one bean, you do not have to select it.
2. Position the mouse pointer over one of the beans you want to copy.
3. Hold down both the Ctrl key and the appropriate mouse button.
   - For Windows, hold mouse button 1.
   - For OS/2, hold mouse button 2.
   - For UNIX platforms, hold mouse button 2.
4. Move the mouse pointer to a new position. To help you with positioning, an outline of the bean appears. When you are copying multiple beans, the outlines of the selected beans move together as a group.
5. When the beans you are copying are in the desired position, release the mouse button and Ctrl key. The copied beans appear where you positioned the outline.

Note: Pressing the Esc key before releasing the mouse button cancels copying without making changes.

Copying a composite bean requires special handling. For information, see “Code Generated from Visually Composed Beans” on page 33.

Deleting Beans

To delete beans, select them and press **Delete** or select **Delete** from the pop-up menu.

When you delete a connected bean, the connections between that bean and other beans are also deleted. However, when you select **Edit** and then **Undo**, you restore the deleted bean and any connections that were removed.

Displaying Bean Pop-Up Menus

To see a menu of operations you can perform on a bean, click mouse button 2 on the bean. The pop-up menu for the bean appears. Choices on the pop-up menu allow you to delete the bean, rename it, and perform other operations (which vary, depending on the bean).

To display a pop-up menu for multiple beans:

- Select the beans.
- Place the mouse pointer over any of the selected beans.
- Click mouse button 2.

Note: When you open a pop-up menu for multiple selected beans, one menu displays the choices common to all selected beans. Operations performed from that pop-up affect all selected beans.

RELATED CONCEPTS

“Setting Tabbing Order” on page 12

“Layout Managers in Visual Composition” on page 14

“Tearing Off Properties” on page 13

“Property-to-Property Connections” on page 28

“Beans Palette” on page 9

RELATED TASKS

“Setting Layout Properties during Visual Composition” on page 60

“Opening the Property Sheet for a Bean” on page 56

“Changing Bean Size and Position” on page 59

RELATED REFERENCES

“The Tool Bar in Visual Composition” on page 191

“Align Left” on page 197

“Align Center” on page 197

“Align Right” on page 197

“Align Top” on page 197

“Align Middle” on page 197

“Align Bottom” on page 197

“Vertically In Bounding Box” on page 204

“Horizontally In Bounding Box” on page 204

“Set Tabbing” on page 206

“Morph Into” on page 205

“Chapter 52. Morph Into” on page 229

“Tear-Off Property” on page 206

Chapter 19. Connecting Beans

In VisualAge, you draw connections between beans to define their interaction. This involves using the mouse to select a feature of the source bean and connect it to the feature of the target bean. The type of feature at the source—property or event—and the type of feature at the target—property, method, or code—determines the type of connection. For example, if the source is an event and the target is a method, the connection is an event-to-method.

If you decide to change the connection behavior of the bean, you can edit or reorder the existing connections without redrawing them.

Note: You can also perform connections within the Beans List window.

Connecting Features to Other Features

To connect two features, follow these steps. The term *source* refers to the where the connection begins and the term *target* refers to where the connection ends.

1. Select the source bean, click mouse button 2, and select **Connect** from the pop-up menu.

   In most cases, a cascade menu appears that displays the names of the most commonly used (or *preferred*) features. If additional features exist that are appropriate for the connection type, **Connectable Features** also appears on the menu. Selecting **Connectable Features** opens a connection window with an expanded list of features, sorted alphabetically and by feature type.

   - If a connection window appears instead of the cascade menu, this means that preferred features have not been assigned for the bean.
   - If the **Connectable Features** selection does not appear on the menu, this means the menu contains all available features, not just the preferred ones, and there are no more from which to select.
2. Select a feature by doing one of the following:
   - If the feature appears in the preferred list, select it.
   - If the feature does not appear in the list but the **Connectable Features** selection is available, select **Connectable Features** and then select the feature from the expanded list in the connection window.
   - If the feature does not appear in either the preferred or expanded list, you may be able to edit the bean to add the feature you need.
3. If, at this point, you decide not to complete the connection, do one of the following:
   - If a pop-up menu appears, move the mouse pointer away from the connection menu and click mouse button 1.
   - If a window showing all the features appears, click **Cancel**.

   The menu or window closes without completing the connection.
4. Place the mouse pointer over the target bean. As you move the mouse, a dashed line trails from the mouse pointer back to the source bean.
5. Click mouse button 1. As with the source bean, either a pop-up menu or connection window appears.
6. Select the target feature as before.

When you complete the connection, a colored connection line appears. The color indicates the connection type, based on the features you selected as end points.

You make connections within the beans list in the same manner as in the Visual Composition Editor. You cannot, however, start a connection on the Visual Composition Editor and complete it in the beans list or vice versa. You may want to draw connections on the beans list instead of the Visual Composition Editor if you use a layout that allows for a bean to completely cover another bean.

Note: If you are using an unbound property in a property-to-property connection, open properties on the connection and select an event to associate with the property. When the event is triggered, the properties values align.

Connecting Features to Code

The source for a code connection must be either an event or a bound property (a property that fires an event when its value changes). Connect to a code as follows:

1. Open the pop-up menu for the source bean.
2. Select **Event to Code** and the Event-to-Code Connection window appears.
3. Select an **Event** from the **Event** drop-down.
4. If you have already written the code, select it from the Method drop-down. Otherwise, leave `<new method>` visible in the Method field.
5. Modify the code in the code window as appropriate. This window operates the same as the code window for creating methods in the IDE.
6. If you want the event to pass its parameters to the new method, select **Pass event data** at the bottom of the panel.
7. If the code takes input parameters and you want to specify them as constants, save the code by opening the code pane pop-up and selecting Save. When the code is saved, click **Set parameters** and enter the constants you want.
8. Click **OK**.

   The connection window closes and VisualAge draws a green connection arrow from the source bean to a moveable text box on the free-form surface. If the connection arrow is dashed, you must supply values for the input parameters of the code.

   You can also create a code connection by selecting a source event and targeting an Event-to-Code Connection on the free-form surface.

Connecting from Connection Results

An *exception* is any user, logic, or system error detected by a function that does not deal with the error itself but passes the error on to a handling routine, called an *exception handler*. In VisualAge, you can catch exceptions by connecting exception events to either methods or code.

An exception is a feature of a connection, not a bean. It appears as *exceptionOccurred* on the connection's connection menu.

You can also pass the return value from the target of a connection. This return value displays as the *normalResult* event of the connection. You can connect the *normalResult* event to a feature of the same bean or another bean. For example, you can connect the exceptionOccurred to a method that brings up a prompter with an error message.

Supplying Parameter Values for Incomplete Connections

Connections sometimes require parameters, or input arguments. If a connection requires parameters that have not been specified explicitly or by default, it appears as a dashed arrow, indicating that it is incomplete. When you have made all the necessary parameter connections, the connection line becomes solid, indicating that the connection is complete.

You can create parameter from method and parameter from code connections. When the primary connection calls for a value parameter, it calls the method or code, which passes the return value as the parameter.

Supplying a Parameter Value Using a Connection

1. Start a new connection using as the source, the dashed connection line that requires the parameter.
2. For the target, select the feature that provides the value.

Supplying a Parameter Value Using a Constant

When connections need parameters with constant input values, provide these values through the properties window of the incomplete connection, as follows:

1. Open properties for the incomplete connection by selecting **Properties** from the pop-up menu or by double-clicking the connection line. The properties window of the incomplete connection appears.
2. Select **Set parameters**. The Constant Parameter Value Properties window appears showing the parameters for which you can set constant values.
3. Enter the constant parameter values you want to use.
4. Do one of the following:
   - To apply and save the values and close the window, click **OK**.
   - To close properties without saving any of the parameter values you just entered, click **Cancel**.

Specifying Values for Parameters by Default

In most connections other than event-to-code, data is not passed by default from the source of a connection to the target. However, you can set VisualAge so that it always passes the available event data. In that case, the initial connection line may not appear dashed.

To set the connection to always pass event data, follow these steps:

1. Open the properties for the connection.
2. Select the **Pass event data** check box.

Note: The source of the connection determines the event data that is passed.

The event-to-code dialog defaults the **Pass event data** check box to false only if either of the following is not true:

- The event passes a parameter and the code accepts a method.
- The event parameter matches the type of the parameter accepted in the code.

Editing Connection Properties

Connection properties enable you to change a connection without redrawing it. Through the properties window, you can do the following:

- Change the source or target feature, depending on the connection type
- Reverse the direction of a property-to-property connection
- Specify an input parameter as a constant
- Delete the connection

To open properties for a connection from either the free-form surface or the Beans List window, move the mouse pointer over the connection and do one of the following:

- Double-click mouse button 1.
- Click mouse button 2 and select **Properties** from the connection’s pop-up menu.

RELATED CONCEPTS

“Parameter Connections” on page 30

“Chapter 8. Connections” on page 27

“Code Connections” on page 29

RELATED TASKS

“Reordering Connections” on page 82

“Selecting and Deselecting Connections” on page 81

“Deleting Connections” on page 81

“Showing and Hiding Connections” on page 81

“Changing the Source and Target of Connections” on page 83

RELATED REFERENCES

“Beans List” on page 196

“Pass Event Data” on page 221

“Chapter 50. Event-to-Code Connection Window” on page 225

Chapter 20. Manipulating Connections

Once you have made connections to and from beans on the free-form surface, you can modify them as follows:

- Display or hide the connection lines
- Delete the connection
- Reorder the connections from a bean
- Change the connection name
- Change the source and target of the connection without starting over

Showing and Hiding Connections

You can show and hide connections by using the **Show connections** and **Hide connections** tools. They can be found on the tool bar or as selections on the **Tools** pull-down menu. These tools show or hide all connections that have the selected bean or beans as their end points. If no beans are selected, these tools show and hide all connections in the composite bean.

If you hide connections, the Visual Composition Editor is refreshed faster and is less cluttered, making it easier for you to work.

You can also show and hide connections from the pop-up menu by selecting the **Browse Connections** cascade menu. The choices in this menu affect only connections going to and from the bean whose pop-up menu you opened.

Deleting Connections

To delete a connection, do one of the following:

- Select the connection and press the **Delete** key.
- From the connection's pop-up menu, select **Delete**.
- From the connection's properties window, click **Delete** .

Selecting and Deselecting Connections

You select connections in the same way that you select beans. When you select a connection, boxes called *selection handles* ■ appear on it to show that it is selected. When first drawn, a connection contains three selection handles: one at each end and one in the middle. You can use selection handles to change either of the following:

- The end points of the connection.
- The shape of the connection line, by dragging the middle box to another location. This helps you distinguish among several connection lines that are close together.

Selecting a Single Connection

1. Move the mouse pointer over the connection you want to select.
2. Click mouse button 1 and the connection is selected.

Selecting Multiple Connections

To select multiple connections, do one of the following:

- For Windows platforms: Hold down Ctrl or Shift and click mouse button 1 on each connection you want to select.
- For UNIX platforms and OS/2: Hold down mouse button 1 and move the mouse pointer over each connection you want to select. After you select the connections, release mouse button 1.

Deselecting Connections

To deselect a connection without selecting another bean or connection, follow these steps:

1. Move the mouse pointer over the connection line.
2. Hold down the Ctrl key.
3. Click mouse button 1.

Reordering Connections

If you make several connections from the same bean, they run in the order in which you made the connections. To ensure the correct flow of control when you generate the source code, you might need to reorder the connections. If so, do the following:

1. Select the source bean.
2. From the source bean pop-up menu, select **Reorder Connections From**. The Reorder Connections window appears, showing a list of your connections.
3. With the mouse pointer over the connection you want to reorder, press and hold the appropriate mouse button:
   - In OS/2, mouse button 2
   - In Windows, mouse button 1
   - In AIX, mouse button 3
4. Drag the connection to the place in the list where you want the connection to occur.

   Note: Parameter connections must always follow the connections they supply.
5. Release the mouse button.
6. Repeat these steps until the connections are listed in the order in which you want them to occur.
7. Close the window.

Changing the Connection Name

You can change the name of a connection to make identification easier. To change the connection name:

1. Open the pop-up for the connection.
2. Select **Change Connection Name**.
3. Modify the connection name.

The connection name changes in the Visual Composition Editor and, after you save the bean, in the source code.

Changing the Source and Target of Connections

You can change the end points of a connection without redrawing it, either by dragging the connection or by changing its properties.

You can change the source of any connection. In most cases, you can also change the target. However, you cannot change the target to a type that is not allowed. For example, you cannot change a target to an event because an event can only be the source of a connection.

Moving Either End of a Connection to a Different Bean

1. Select the connection.
2. Move the mouse pointer over the appropriate selection handle at the end of the connection.
3. Press and hold mouse button 1.
4. Move the mouse pointer to the new bean or connection.
5. Release the mouse button.

If you change the target of a *feature*-to-method connection to a bean that does not support the target method, the connection menu appears, and you can select a new target feature.

Moving Either End of a Connection to a Different Feature

1. Open properties for the connection. The **Properties** window for that connection type appears.
2. Select new end points from the lists shown.
3. Click **OK**.

Reversing the Direction of a Connection

The direction of property-to-property connections determines which end point is initialized first. The target property is initialized first based on the value of the source. Only property-to-property connections can be reversed. To do this, open properties for the connection and select the **Reverse** push button.

Changing the Shape of a Connection

To help you distinguish among several connection lines that are close together, you can change the shape of connections. To do this, follow these steps:

- Select the connection line you want to change.
- Place the mouse pointer over the middle selection handle of the connection line.
- Click and hold mouse button 1 and drag the connection line to the desired shape.
- Release the mouse button and the new line is set with two new midpoint handles.

You can change the connection back to its original shape by selecting **Restore Shape** from the pop-up window.

RELATED CONCEPTS

“Chapter 8. Connections” on page 27

RELATED TASKS

"Selecting and Deselecting Beans" on page 71

"Chapter 19. Connecting Beans" on page 77

"Editing Connection Properties" on page 80

"Displaying Bean Pop-Up Menus" on page 75

RELATED REFERENCES

"Show Connections" on page 196

"Hide Connections" on page 196

"Reorder Connections From" on page 205

"Change Connection Name" on page 202

Chapter 21. Managing the Beans Palette

You can modify the beans palette at any time from the Visual Composition Editor and in any of the following ways:

- Add a bean
- Add a category
- Add a grouping separator line
- Change icon size
- Refresh the palette
- Remove a bean
- Remove a category
- Reorder beans
- Resize the palette

To resize the palette, drag the sizing handle on the right side of the palette. If you choose not to resize the palette and some of the beans are not visible, you can access the beans by using the scroll buttons that appear at the top and bottom of the palette.

To change the icon size on the palette and the Beans List, open the palette pop-up and select **Show Large Icons.** This is a toggle option with the default set to small icons (16x16). The large icons are 32x32.

To reorder the beans within a category, or move a bean to another category:

1. From the palette pop-up or the **Bean** pull-down, select **Modify Palette.**
2. From the Palette list, drag the bean to the position or category you desire.
3. Click **OK**.

If you have a category with many beans, you can group common beans using separators. To add a grouping separator to a category:

1. Select the category.
2. Select Add Separator and VisualAge adds a separator to the end of the category.
3. Select the new separator line and drag it to the desired location.

To remove a separator line, select the separator and then click **Remove.**

You may need to refresh the palette to view the following changes:

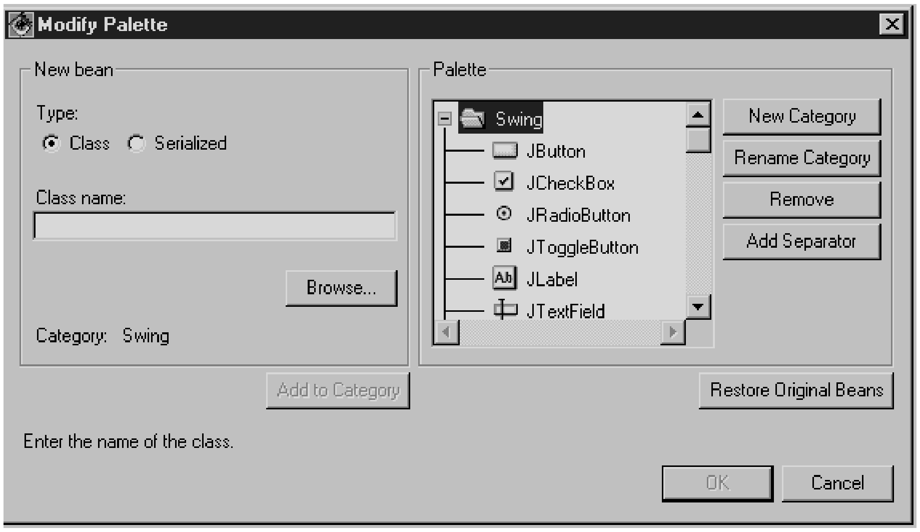
- Changing the icon that represents a bean
- Adding new beans
- Loading installed features manually at the package or class level

To refresh the palette, select **Refresh Palette** from the palette pop-up.

Adding a Category to the Palette

1. From the palette pop-up, select **Modify Palette**, and the Modify Palette window appears.
2. From the Palette group box, select **New Category** and a new category item appears highlighted in the list.
3. Enter the name for your new category in the highlighted region.

4. Enter or click **OK**.



Type: ● Class ○ Serialized

Class name:

Category: Swing

Enter the name of the class.

RELATED CONCEPTS

"Beans Palette" on page 9

"Chapter 13. Object Serialization in VisualAge" on page 47

RELATED TASKS

"Chapter 17. Composing Beans Visually" on page 67

RELATED REFERENCES

"Chapter 42. Modify Palette Window" on page 209

"Refresh Palette" on page 205

Adding a Bean to the Palette

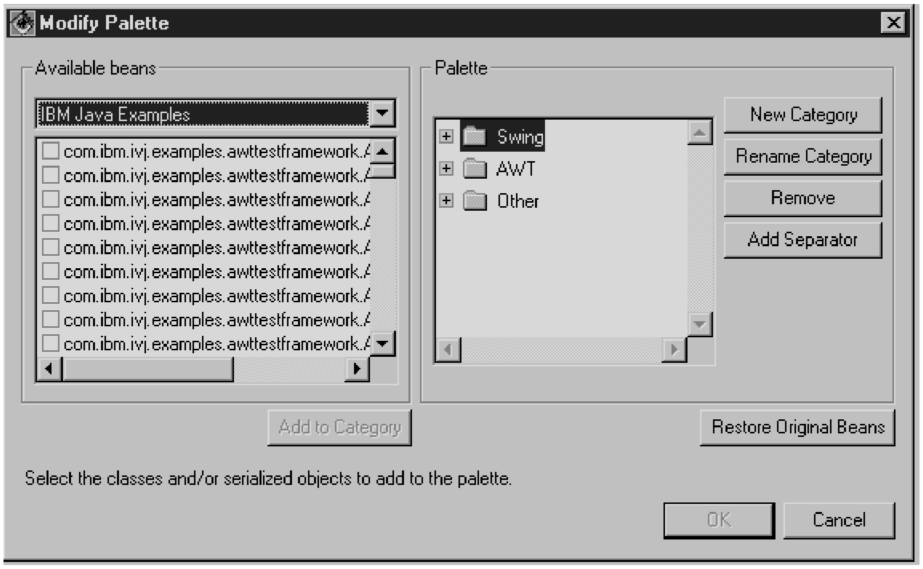
To add beans to any category on the beans palette:

1. From the palette pop-up, select **Modify Palette**, and the Modify Palette window appears.
2. Select the bean type.
   - If you selected the Class bean type, enter the class name in the entry field or select **Browse** to locate the class.
   - If you selected the Serialized bean type, enter the file name in the entry field or select **Browse** to locate the serialized object file to add.
3. Under **Palette**, select a category for the new bean.
4. Select **Add to Category** and click **OK**. The Visual Composition Editor adds the bean to the category on the beans palette.

To add beans from a project to any category on the beans palette:

1. From the palette pop-up, select **Add Bean from Project**, and the Modify Palette window appears.
2. From the **Available beans** pane, open the project in the drop-down list and select the project that contains the beans you want to add to the palette.

3. Select the beans, by selecting the check boxes.
4. Under **Palette**, select a category for the new beans.
5. Select **Add to Category** and click **OK**. The Visual Composition Editor adds the beans to the category on the beans palette.



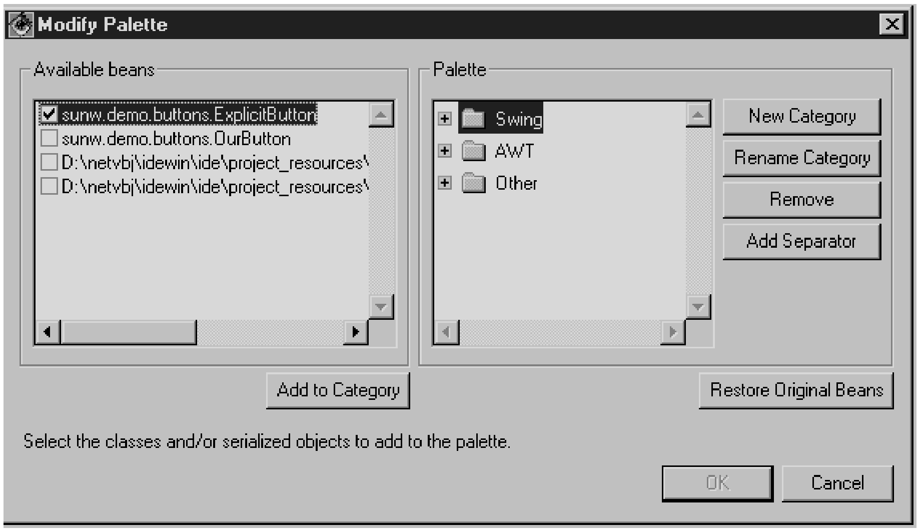
IBM Java Examples

□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A
□ com.ibm.ivj.examples.awttestframework.A

⊞ Swing
⊞ AWT
⊞ Other

Select the classes and/or serialized objects to add to the palette.

You can also add class or serialized files from .jar files to the palette by following the import SmartGuide from the **File** menu. The Modify Palette window that appears contains an **Available beans** list, where you select the beans you want to add to the category you select, or create, in the **Palette** list.



☑ sunw.demo.buttons.ExplicitButton
□ sunw.demo.buttons.OurButton
□ D:\netvbj\idewin\ide\project_resources\
□ D:\netvbj\idewin\ide\project_resources\

⊞ Swing
⊞ AWT
⊞ Other

Select the classes and/or serialized objects to add to the palette.

Note: If you designate an icon for the bean in the Information pane of the BeanInfo class, it is used for the palette entry. Otherwise, a default icon is used.

RELATED CONCEPTS

“Beans Palette” on page 9

RELATED TASKS

“Deleting a Bean or Category from the Palette”

RELATED REFERENCES

“Chapter 42. Modify Palette Window” on page 209

“Chapter 56. BeanInfo Page” on page 237

“Add Bean from Project” on page 202

Deleting a Bean or Category from the Palette

To remove a bean from the beans palette:

1. From the **Bean** menu, select **Modify Palette** and the Modify Palette window appears.
2. In the Palette tree view, expand the category that contains the part you wish to remove.
3. Select **Remove** and a confirmation dialog appears.
4. Select **Yes** and then **OK**. The selected bean is removed from the beans palette.

To remove a category from the beans palette:

1. From the **Bean** menu, select **Modify Palette** and the Modify Palette window appears.
2. In the list box, select the category that you wish to remove.
3. Select **Remove** and **OK**. The selected category is removed from the beans palette.

RELATED CONCEPTS

“Beans Palette” on page 9

RELATED TASKS

“Adding a Category to the Palette” on page 85

“Adding a Bean to the Palette” on page 86

RELATED REFERENCES

“Chapter 42. Modify Palette Window” on page 209

Chapter 22. Using VisualAge Beans in Visual Composition

You can use VisualAge beans, property settings and connections to compose a wide variety of Java program elements. The following topics explain how to use VisualAge beans to compose program elements in the Visual Composition Editor.

Composing with User Interface Beans

VisualAge provides a set of user interface beans that you can use to compose an applet or application. The following topics explain how to use these beans:

- “Composing an Applet”
- “Composing a Window” on page 92
- “Adding a Pane or Panel” on page 95
- “Adding a Table or Tree View” on page 99
- “Adding a Text Component” on page 101
- “Adding a List or Slider Component” on page 104
- “Adding a Button Component” on page 107
- “Adding a Menu or Tool Bar” on page 109

Composing with Factory and Variable Beans

VisualAge provides Factory and Variable beans that you can use to create and access bean instances. The following topic explains how to use these beans:

- “Dynamically Creating and Accessing a Bean Instance” on page 113

RELATED CONCEPTS

“Chapter 2. How Classes and Beans Are Related” on page 3

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

RELATED REFERENCES

“Chapter 28. Beans for Visual Composition” on page 141

Composing an Applet

Applets are programs that can be downloaded and run by a Java-enabled web browser. These programs are generally small and specialized. An applet runs in a web page on a client system, within bounds specified by the page markup. A Java applet operates within constraints that provide security from remote system intrusion.

You can compose and test an applet in the Visual Composition Editor. To run an applet in a web page, export the applet class and edit the web page source file to include the applet.

VisualAge provides applet beans from the *com.sun.java.swing* and *java.applet* packages, as well as others. Although Swing and AWT beans can be mixed, it is inadvisable.

1. Create one of the following applet beans:

Bean	Description
"JApplet" on page 144 or "Applet" on page 143	A program that can run in a compatible web browser

   Creating an applet bean

   Create an applet as a new composite bean. You can create the bean in any of the following ways:

   - From the Quick Start window. Select **Quick Start** from the **File** pull-down menu to open the Quick Start window. Select *Basic* in the left pane, *Create Applet* in the right pane, and the **OK** button to open the SmartGuide – Create Applet window. See the related task topic on Creating an applet for details.
   - From the **Create Applet** tool bar button. Select the A button on the tool bar to open the SmartGuide – Create Applet window. Specify one of the applet beans as the superclass for your applet, and request to compose the class visually. See the related task topic on Creating an applet for details.
   - From the **Create Class** tool bar button. Select the C button on the tool bar to open the SmartGuide – Create Class window. Specify one of the applet beans as the superclass for your applet, and request to compose the class visually. See the related task topic on Creating a class for details.

2. Compose the user interface and logic for the applet. Add and arrange visual components, add nonvisual beans, and connect them to establish functional relationships.

   Accessing a JApplet bean in the Visual Composition Editor

   When you create a JApplet bean, a content pane is also added in which to place other components. With the exception of a JMenuBar, user interface components are added to the content pane. As a result, the content pane completely covers the Japplet bean in the Visual Composition Editor. To access the JApplet bean, open the Beans List.

   Resizing or moving a JApplet bean in the Visual Composition Editor

   Select the JApplet bean in the Beans List. Then, you can resize or move the JApplet in the Visual Composition Editor. If you try to select the JApplet bean in the Visual Composition Editor, you will select its content pane instead.

   Replacing the content pane for a JApplet bean

   To replace the content pane, delete it and add another container component. When you delete the content pane, a warning is displayed indicating that the content pane is missing. If you save the bean without adding another content pane, a JPanel bean is used.

   Arranging beans in an applet

   Use either of the following methods:

- Use a layout manager to control size and position of beans within the applet or applet content pane.
- Without using a layout manager, place beans approximately where you want them and use visual composition tools to align them.

Accessing the applet context

The applet context represents the environment in which an applet is running. It provides methods to get an image from a URL, to get an audio clip from a URL, to find other applets within the document, and to show a document at another URL. To access these applet context methods, tear off the *appletContext* property of the Applet or JApplet bean. Access a JApplet bean in the Beans List.

Accessing the document or applet URL

To get the URL of the HTML file that the applet is running in, connect to the *documentBase* property or the *getDocumentBase* method of the Applet or JApplet bean. To get the URL of the applet, connect to the *codeBase* property or the *getCodeBase* method of the Applet or JApplet bean.

Providing information about the applet

To define information about your applet, edit the *getAppletInfo* method on the Methods page. To get the applet information for an About dialog, connect to the *appletInfo* property or the *getAppletInfo* method of the Applet or JApplet bean.

3. Test the applet. You can run the applet from the Visual Composition Editor. This makes it easy to iteratively modify and test the applet while you compose it.

   Testing an applet from the Visual Composition Editor

   Do either of the following to test the applet:

   - Select the 🏃 button on the tool bar.
   - From the **Bean** pull-down menu, select **Run**, then either **In Applet Viewer** or **Run main**.

   VisualAge performs the following steps:

   a. Saves the edit description of the bean

   b. Generates source code

   c. Compiles the class

   d. Runs the applet

4. Export the applet class and related resources from VisualAge.

   Exporting an applet class

   Select the applet class in the Workbench window. Then, select **Export** from the **File** menu to open the SmartGuide – Export window. See the related task topic on Exporting a class for details.

5. Edit your web page source file to add the applet. Specify the applet at the location in your page markup where you want the applet to run. If you had VisualAge generate an HTML file when you exported the applet, you need to edit the source to specify attributes for the `<applet>` tag.

   Adding an applet in an HTML file

   Use the `<applet>` tag to identify the applet class and to specify the dimensions of the bounding rectangle in which the applet is to run. The following example includes an applet named MyApplet. The applet is run in a rectangle that is 100 pixels wide and 80 pixels high.

```
<applet code="MyApplet.class" width=100 height=80></applet>
```

See books documenting HTML for other attributes that can be specified on the Applet tag.

For examples, see the *BookmarkList* class in the *com.ibm.ivj.examples.vc.swing.bookmarklist* and *com.ibm.ivj.examples.vc.bookmarklist* packages. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Creating an applet

Creating a class

Exporting a class

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 29. Applet Beans" on page 143

BookmarkList Sample

Composing a Window

Windows beans are the primary visual context for other user interface components. VisualAge provides window beans from the *com.sun.java.swing* and *java.awt* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

You can compose and test a window in the Visual Composition Editor. You should create a new composite bean as a subclass of a window class. You can also add window beans for secondary windows. A FileDialog cannot be composed as a primary window bean. It represents a system file dialog.

1. Create one of the following window beans:

Bean	Description
"JDialog" on page 147 or "Dialog" on page 145	A custom dialog, typically a secondary window
"JFrame" on page 148 or "Frame" on page 146	A desktop window with a title bar, sizing borders, and sizing buttons
"JInternalFrame" on page 148	A frame that is a child of another Swing component
"JWindow" on page 149 or "Window" on page 150	A window without a title bar, sizing borders, and sizing buttons

Creating a window bean

You can create a composite window bean in any of the following ways:

- From the Quick Start window. Select **Quick Start** from the **File** pull-down menu to open the Quick Start window. Select *Basic* in the left pane, *Create Class* in the right pane, and the **OK** button to open the SmartGuide – Create Class window. Select one of the window beans as the superclass for your window, and request to compose the class visually. See the related task topic on Creating a class for details.

- From the **Create Class** tool bar button. Select the C button on the tool bar to open the SmartGuide – Create Class window. Select one of the window beans as the superclass for your window, and request to compose the class visually. See the related task topic on Creating a class for details.

Adding a secondary window to a composite bean
Add a window to the free-form surface of the composite bean.

- For a static window, add one of the window beans.
- For a dynamically created window, add a Factory. From the Factory's pop-up menu, select **Change type**. Then, specify the window bean as the object type. Create the window when an event occurs by connecting the event to a Factory constructor method for the window.

2. Compose the user interface and logic for the window. Add and arrange visual components, add nonvisual beans, and connect them to establish functional relationships.

Defining a window title
Enter text for the *title* property in the window's property sheet.

Accessing a JWindow bean in the Visual Composition Editor
When you create a JDialog, JFrame, or JWindow bean, a content pane is also added in which to place other components. With the exception of a JMenuBar, user interface components are added to the content pane. Because a JWindow has no frame, the content pane completely covers the JWindow bean in the Visual Composition Editor. To access the JWindow bean, open the Beans List.

Resizing or moving a JWindow bean in the Visual Composition Editor
Select the JWindow bean in the Beans List. Then, you can resize or move the JWindow in the Visual Composition Editor. If you try to select the JWindow bean in the Visual Composition Editor, you will select its content pane instead.

Replacing the content pane for a JDialog, JFrame, or JWindow bean
To replace the content pane, delete it and add another container component. When you delete the content pane, a warning is displayed indicating that the content pane is missing. If you save the bean without adding another content pane, a JPanel bean is used.

Arranging beans in a window
Use either of the following methods:

- Use a layout manager to control size and position of beans within the window content pane or client component.
- Without using a layout manager, place beans approximately where you want them and use visual composition tools to align them.

Keeping a dialog in focus until it is closed
Set the *modal* property to *True* in the dialog's property sheet.

Preventing window resizing
Set the *resizable* property to *False* in the window's property sheet.

Opening a window
Connect an event, such as the *actionPerformed* event of a button or menu item, to the window's *show()* method.

Closing a window

Connect an event, such as the *actionPerformed* event of a button or menu item, to the window's *dispose()* method.

Specifying open or save for a file dialog

Specify the file operation for the *mode* property in the dialog's property sheet. Select *LOAD* for an open dialog, or*SAVE* for a save dialog.

Defining initial selection information for a file dialog

You can specify an initial directory, file, or both in the dialog's property sheet. Specify an initial directory name for the *directory* property. Specify an initial file name for the *file* property.

Obtaining information from a dialog when it is closed

Connect the *normalResult* of the *show()* connection to the target property for the information. Then, connect the dialog property that contains the information to the appropriate parameter of the *normalResult*-to-target connection.

For example, if you want to open a dialog to prompt for a name in a text field and return it to a label in the primary window, do the following:

a. Connect an event in the primary window to the *show()* method of the dialog.

b. Connect the *normalResult* of the *show()* connection to the *text* property of the label in the primary window.

c. Connect the *text* property of the text field in the dialog to the *value* parameter of the *normalResult*-to-*text* connection.

3. Test the window. You can run the window from the Visual Composition Editor. This makes it easy to iteratively modify and test the window while you compose it.

Testing a window from the Visual Composition Editor

You can test a composite bean that is a subclass of a window bean in the Visual Composition Editor. Do either of the following to test the window:

- Select the 🏃 button on the tool bar.
- From the **Bean** pull-down menu, select **Run**, then **Run main**.

VisualAge performs the following steps:

a. Saves the edit description of the bean

b. Generates source code

c. Compiles the class

d. Runs the window

For examples, see the *CustomerInfo* class in the *com.ibm.ivj.examples.vc.customerinfo* package. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Creating a class

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 30. Window Beans" on page 145

CustomerInfo Sample

Adding a Pane or Panel

A pane or panel is a container for other components. It is used within another pane or panel, within a window, or within an applet. VisualAge provides pane and panel beans from the *com.sun.java.swing* and *java.awt* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

You can add a pane or panel bean as an embedded container for other components. You can also create a bean as a subclass of one of these beans to define a reusable component. This is particularly useful for panels.

1. Add or create one of the following pane or panel beans:

Bean	Description
"JDesktopPane" on page 151	A pane for a desktop within another Swing container
"JEditorPane" on page 152	A pane for editing defined text types, such as HTML
"JOptionPane" on page 152	A simple dialog pane
"JPanel" on page 153 or "Panel" on page 156	A composition surface for user interface components
"JScrollPane" on page 153 or "ScrollPane" on page 156	A scrollable view for another component
"JSplitPane" on page 154	A split view for other components
"JTabbedPane" on page 155	A tabbed view for other components
"JTextPane" on page 155	A pane for editing text with visible styles and embedded objects

2. Define initial characteristics of the component.

Specifying when to display scroll bars in a JScrollPane or ScrollPane bean
Select a scroll bar policy in the pane's property sheet. For a JScrollPane bean, set horizontal and vertical scroll bar policies with the *horizontalScrollBarPolicy* and *verticalScrollBarPolicy* properties. For a ScrollPane bean, set a policy for both scroll bars with the *scrollBarDisplayPolicy* property. Select one of the following choices to specify when to display the scroll bar or scroll bars:

- *ALWAYS* or *SCROLLBARS_ALWAYS*—always display, regardless of the relative size of the scroll pane and the component it contains.
- *AS_NEEDED* or *SCROLLBARS_AS_NEEDED*—display only when the scroll pane is smaller than the component it contains. For a JScrollPane bean, also specify a preferred size for the *preferredSize* property.
- *NEVER* or *SCROLLBARS_NEVER*—never display, regardless of the relative size of the scroll pane and the component it contains

Enabling scrolling for a null layout in a JScrollPane or ScrollPane bean
If you use a null layout for a panel that is embedded in a scroll pane, set the panel's *size* and *preferredSize* properties to support scrolling by the scroll pane.

Scrolling a JScrollPane bean in the Visual Composition Editor

You can manipulate the scroll bars in a JScrollPane during composition, but not by dragging the scroll box (thumb)

Defining the component orientation in a JSplitPane bean

Set the *orientation* property in the pane's property sheet to arrange components within the split pane. The default setting, *HORIZONTAL*, arranges components on the left and right. To arrange components on the top and bottom, select *VERTICAL*.

Adding components to a JSplitPane bean

You can add two components to a split pane. When you select a component to add, hold down the mouse button to drag it over one half of the split pane. When the mouse pointer is over the split pane and before you release the mouse button, a target outline appears around the half of the split pane that would receive the component. Drop one component on each half of the split pane. Before you add the second component, the first component appears to fill the split pane, but the target outline appears when you move the mouse pointer over the side of the split pane that does not yet have a component.

Defining the divider for a JSplitPane bean

Set the *dividerLocation* property to specify the initial position of the divider between the panes. This has no effect unless two components have been added to the split pane. Set the *dividerWidth* property to specify the initial width of the divider. Set the *oneTouchExpandable* property to *True* to enable the user to adjust the width of the divider.

Composing the first tab page of a JTabbedPane bean

When you drop a JTabbedPane, VisualAge automatically adds a JPanel and names it Page. Customize this page to serve your purpose. Change the tab and add the components you want. If you do not want a JPanel as the page component, delete it and add another component. To avoid background paint problems when you delete a page, set the *opaque* property of the JTabbedPane to *True*.

Adding a tab page to a JTabbedPane bean

Drop the tab page component you want on the tab region of the pane. If you drop the component on a tab, a tab containing the new component is inserted after the tab you dropped the component on. If you drop the component after the last tab in the tab region, the new component is added as the last tab page.

Switching the composition focus to a tab page component

Select a tab page component to work with in any of the following ways:

- Select on the tab with a mouse. Then, click on the tab page component to shift the focus from the tabbed pane to the tab page component.
- Select the tab page component in the Beans List window.

Defining the tab for a JTabbedPane page

Define the tab in the property sheet as follows:

- Specify the tab text you want for the *tabTitle* property.
- By default, tabs are at the top of the pane. If you want the tabs on a different edge of the pane, select the edge you want in the *tabPlacement* property.

- If you want an icon on the tab, select an icon for the *tabIcon* property. If the tab can be disabled, you should also specify an icon for the *tabDisabledIcon* property.
- If you want tool tip text for the tab, specify the text for the *tabTip* property. Tool tip text is useful for expanding on the tab title to explain what what a tab page provides.
- To change tab colors, set the *tabBackground* and *tabForeground* properties.
- To specify whether the tab page is initially enabled or not, set the *tabEnabled* property.

Composing minor tabs in a JTabbedPane bean

Add a JTabbedPane bean as a tab component in the primary tabbed pane. Then, define tab placement for the minor tabs on a different edge of the nested tabbed pane.

Customizing a JOptionPane dialog

For standard dialogs, you can call one of the JOptionPane static methods without adding a JOptionPane bean. These methods are described in the task on opening a standard JOptionPane dialog. If you want to customize a dialog, add a JOptionPane bean as follows:

a. Add a JDialog or JInternalFrame bean as the frame for the option pane.

b. Delete the content pane of the frame bean.

c. Add the JOptionPane as the content pane for the frame bean.

Then, set customized properties in the pane's property sheet, including the following:

- Specify the message type. Select one of the following choices for the *messageType* property to specify the nature of the message:
  - *ERROR_MESSAGE*
  - *INFORMATION_MESSAGE*
  - *WARNING_MESSAGE*
  - *QUESTION_MESSAGE*
  - *PLAIN_MESSAGE*
- Specify the option selection. Select one of the following choices for the *optionType* property to specify the button choices for the dialog:
  - *DEFAULT_OPTION*
  - *YES_NO_OPTION*
  - *YES_NO_CANCEL_OPTION*
  - *OK_CANCEL_OPTION*
- To specify an initial option selection, set the *initialValue* property
- To provide an initial value for prompted input, set the *initialSelectionValue* property
- To specify an icon, set the *icon* property

Defining initial properties of a JEditorPane bean

Define initial properties in the pane's property sheet, including the following:

- Specify the text content type for the *contentType* property. For example, you can specify one of the following:
  - *text/plain*—uses the DefaultEditorKit

- *text/html*—uses the HTMLEditorKit
- *text/rtf*—uses the RTFEditorKit
- *application/rtf*—uses the RTFEditorKit

- Enter any initial text for the *text* property.
- For HTML, you can specify a document page instead of initial text. Specify the URL as a quoted string for the *page* property.

Specifying a styled document for a JTextPane bean

Set the *styledDocument* property in the pane's property sheet.

Arranging beans in a JPanel or Panel bean

Use either of the following methods:

- Use a layout manager to control size and position for beans within the panel
- Without using a layout manager, place beans approximately where you want them and use visual composition tools to align them

Defining tool tip text

For Swing components, you can specify tool tip text, also known as fly-over text or hover help. Enter text for the *toolTipText* property in the component's property sheet.

Defining initial availability

By default, the component is initially enabled for user interaction. To initially disable the component, set the *enabled* property to *False* in the component's property sheet.

3. Provide runtime logic for the component.

Opening a standard JOptionPane dialog

The JOptionPane class provides a set of static methods for standard dialogs. These methods have several signatures, enabling you to specify certain dialog characteristics. Call any of these methods by creating an event-to-code connection and specifying the method as the target.

Standard dialog type	For a dialog frame, use ...	For an internal frame, use ...
Confirmation dialog	showConfirmDialog()	showInternalConfirmDialog()
Input dialog	showInputDialog()	showInternalInputDialog()
Message dialog	showMessageDialog()	showInternalMessageDialog()
General dialog with all of the preceding elements	showOptionDialog()	showInternalOptionDialog()

The code should process any selected options or requested input. Depending on the dialog type, the following options can be selected by the user and returned from the dialog:

- *YES_OPTION*—the user selected the **Yes** button
- *NO_OPTION*—the user selected the **No** button
- *CANCEL_OPTION*—the user selected the **Cancel** button
- *OK_OPTION*—the user selected the **OK** button
- *CLOSED_OPTION*—the user explicitly closed the frame

Opening and closing a customized JOptionPane dialog

If you add a JOptionPane bean for customization, process the dialog as follows:

a. Open the pane's frame to display the dialog. Connect an event to the frame's *show()* method.

b. Close the pane's frame when the user selects an option to close the dialog. Connect the pane's *propertyChange* event to the frame's *setVisible()* method. Then, set the connection parameter to *False*.

c. Connect the pane's *propertyChange* event to a property to retrieve an option or input value. Connect the parameter for the event-to-property connection to one of the following option pane properties:

- *value*—the selected option
- *inputValue*—the requested input value

Disabling or enabling a component

Connect a related event to the button's *enabled* property. Specify the parameter value for this connection in one of the following ways:

- To disable the component, open the connection's properties and set the parameter value to *False*.
- To enable the component, open the connection's properties and set the parameter value to *True*.
- To set the new state from another Boolean property, connect the parameter to the other Boolean property.

For examples, see the *CustomerInfo*, *AddressView*, and *CustomerView* classes in the *com.ibm.ivj.examples.vc.customerinfo* package, and the *DirectoryExplorer* class in the *com.ibm.ivj.examples.vc.swing.directoryexplorer* package. The *AddressView* and *CustomerView* classes subclass a Panel as a reusable bean. The *showMessageBox()* method of the *CustomerInfo* class uses a JOptionPane. The *DirectoryExplorer* class uses a JSplitPane. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 31. Pane and Panel Beans" on page 151

CustomerInfo Sample

Adding a Table or Tree View

A table or tree provides a view of objects from a data model that organizes objects in a tabular or expandable tree format. VisualAge provides table and tree beans from the *com.sun.java.swing* and *com.sun.java.swing.table* packages. You should not use these beans with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

1. Add one of the following beans:

Bean	Description
"JTable" on page 159	A table view of objects from a table data model
"JTree" on page 160	A tree view of objects from a tree data model

2. Define initial characteristics of the component.

Defining tool tip text

For Swing components, you can specify tool tip text, also known as fly-over text or hover help. Enter text for the *toolTipText* property in the component's property sheet. To provide tool tip text for table cells, set the *toolTipText* property of the cell renderer.

Defining initial availability

By default, the component is initially enabled for user interaction. To initially disable the component, set the *enabled* property to *False* in the component's property sheet.

3. Add a data model for the component.

Defining a table data model

Create a data model class as a subclass of the AbstractTableModel class. The AbstractTableModel provides most of the TableModel interface, but you will need to implement the following methods:

getRowCount()	By default, this method returns 0. You should return the number of rows in the data array.
getColumnCount()	By default, this method returns 0. You should return the number of columns in the data array.
getValueAt(int, int)	By default, this method returns null. You should return the data array object at the row and column specified by the arguments.

You will also need to provide column names and row data. You can do this in the Methods page by by creating a field of column names as an array of Strings whose initial value is names. You can populate the data model with data that is either fixed or dynamically derived. You could create a field for row data as a two-dimensional array of Objects, or as a Vector.

To use a data model class in the Visual Composition Editor as the data model for a JTable, add the class to the free-form surface. Then, connect the data model's *this* property to the table's *model* property.

Defining table columns

By default, a table uses all columns for each row in the data model. If you want to display data in a subset of columns or reorder the columns, add a TableColumn bean to the table for each column you want to use. Map the table column to the data model column by specifying the 0-based index of the model column for the *modelIndex* property in the TableColumn's property sheet. Customize the column heading by specifying a value for the *headerValue* property.

Defining a tree data model

Create a data model class as a subclass of the DefaultTreeModel class. Define the tree nodes in the data model class.

To use a data model class in the Visual Composition Editor as the data model for a JTrele, add the class to the free-form surface. Then, connect the data model's *this* property to the tree's *model* property.

4. Provide runtime logic for the component.

Getting a selection from a table

- To get the selected row, make a connection from the table's *selectedRow* property.

- To get the selected column, make a connection from the table's *selectedColumn* property.
- To get the contents of a selected cell, connect the table's *getValueAt()* method to the target property or parameter. Then, connect the *selectedRow* property to the first parameter of the *getValueAt()* connection, and the *selectedColumn* property to the second parameter.

Disabling or enabling a component

Connect a related event to the button's *enabled* property. Specify the parameter value for this connection in one of the following ways:

- To disable the component, open the connection's properties and set the parameter value to *False*.
- To enable the component, open the connection's properties and set the parameter value to *True*.
- To set the new state from another Boolean property, connect the parameter to the other Boolean property.

For examples, see the *DirectoryExplorer* class in the *com.ibm.ivj.examples.vc.swing.directoryexplorer* package and the *Amortization* class in the *com.ibm.ivj.examples.vc.swing.mortgageamortizer* package. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 32. Table and Tree Beans" on page 159

MortgageAmortizor Sample

Adding a Text Component

Text components are available for simple text and for enhanced text and editing panes. VisualAge provides text beans from the *com.sun.java.swing*, *com.sun.java.swing.text*, and *java.awt* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

You can add a text bean to enable text input or provide a label.

1. Add one of the following text beans:

Bean	Description
"JLabel" on page 161 or "Label" on page 163	A label, usually to identify another component
"JPasswordField" on page 162	A text field for sensitive data
"JTextArea" on page 162 or "TextArea" on page 164	A multiline text area
"JTextField" on page 163 or "TextField" on page 164	A single-line text field

2. Define initial characteristics of the text component.

Defining label text or initial input text

Enter text for the *text* property in the text component’s property sheet. For a multiline text area, use *\n* to denote a new line.

Adding a label graphic

For a JLabel bean, you can add a graphic to the label. Select a graphic file for the *icon* property in the label’s property sheet. Use the *horizontalTextPosition* property to specify the position of text relative to the graphic. The position choices are *LEFT*, *CENTER*, and *RIGHT*. The default choice is *RIGHT*. Set the *iconTextGap* property to specify the space between the icon and label text.

Defining keyboard access to an input field

For Swing text input components, you can define keyboard shortcuts to place the focus in the input field or area.

- To define an accelerator for a text field, specify the accelerator character, enclosed in single quotes, for the *focusAccelerator* property in the text component’s property sheet. If you define tool tip text for the text field, the accelerator is displayed with the tool tip text. For example, if you specify *’a’* as the *focusAccelerator* value, *alt+A* appears after tool tip text when the user moves the mouse pointer over the fext field.
- To define a label mnemonic for a text field, specify the mnemonic character, enclosed in single quotes, for the *displayedMnemonic* property in the JLabel’s property sheet.

Aligning text

Select an alignment choice for the *horizontalAlignment* or *alignment* property in the text component’s property sheet. The alignment choices are *LEFT*, *CENTER*, and *RIGHT*.

Selecting initial text

To select the initial text, set the *selectionStart* and *selectionEnd* properties in the text component’s property sheet. These values are offsets from the beginning of the text, which is at offset 0. To select all text without determining the initial text length, specify selection from offset 0 to an offset that you consider to be larger than the initial text length.

Defining a minimum size for layout managers

Some layout managers use a minimum size for placement of components. To specify a minimum width for a text area, enter the width, in characters, for the *columns* property in the text component’s property sheet. To specify a minimum height for a text area, enter the number of rows for the *rows* property.

Hiding input text

To hide input text, either use a JPasswordField bean or specify an echo character for the *echoChar* property in the text component’s property sheet.

Preventing text modification in a text area

To prevent any input in a text area, set the *editable* property to *False* in the text component’s property sheet.

Making a text area scrollable

An AWT TextArea implements scrolling, duplicating the capability of a ScrollPane. A JTextArea does not implement scrolling itself, but uses the scrolling capability of a JScrollPane in which it is placed. If you want a JTextArea to be scrollable, drop it in a JScrollPane.

Defining tool tip text for a text component
For Swing components, you can specify tool tip text, also known as fly-over text or hover help. Enter text for the *toolTipText* property in the component's property sheet.

Defining initial availability
By default, the component is initially enabled for user interaction. To initially disable the component, set the *enabled* property to *False* in the component's property sheet.

3. Provide runtime logic for the text component.

Synchronizing text
If you need to synchronize user input between two text components, do the following:

a. Connect their *text* properties.
b. Open the connection properties.
c. Associate each end of the connection with the keyReleased(java.awt.event.KeyEvent) event.

If you need to synchronize text that you set by connection from another source, make connections to both text components.

Disabling a text component
To disable a text component when an event occurs, connect the event to the component's *enabled* property and set the connection parameter value to *False*.

Enabling a text component
To enable a text component when an event occurs, connect the event to the component's *enabled* property and set the connection parameter value to *True*.

For examples, see the *LayoutManagers* class in the *com.ibm.ivj.examples.vc.swing.layoutmanagers* and *com.ibm.ivj.examples.vc.layoutmanagers* packages and the *Amortization* class in the *com.ibm.ivj.examples.vc.swing.mortgageamortizer* and *com.ibm.ivj.examples.vc.mortgageamortizer* packages. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

LayoutManagers Sample

MortgageAmortizor Sample

Adding a List or Slider Component

List components provide a list of items for the user to select. Slider components show a range of selection values or show progress for the duration of an operation. VisualAge provides list and slider beans from the *com.sun.java.swing* and *java.awt* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

You can add a list or slider bean to enable the user to select an item or value.

1. Add one of the following list or slider beans:

Bean	Description
"JComboBox" on page 168 or "Choice" on page 167	A selectable list with an entry field
"JList" on page 168 or "List" on page 170	A selectable list of choices
"JProgressBar" on page 169	A progress indicator
"JScrollBar" on page 169 or "Scrollbar" on page 171	A scrolling component
"JSlider" on page 170	A selection component for a range of values

2. Define initial characteristics of the component.

Making a JList bean scrollable
An AWT List implements scrolling, duplicating the capability of a ScrollPane. A JList does not implement scrolling itself, but uses the scrolling capability of a JScrollPane in which it is placed. If you want a JList to be scrollable, drop it in a JScrollPane.

Defining choices for a JComboBox, Choice, JList, or List bean
For all of these components, you can specify choices using an initialization method. For a JList, you can alternatively define the list in a ListModel. Add the list model class to the free-form surface. Then, connect the list model's *this* property to the JList's *model* property.

To specify the choices using an initialization method, follow these steps:

a. After adding the bean, note its name. If you select the bean, its name appears in the Visual Composition Editor status area.

b. Save your composite bean.

c. On the Methods page, add a method to initialize the choices. The method signature should look like this:

```
void initializeChoices(choiceType myChoices);
```

Specify the appropriate class for the *choiceType*:

- com.sun.java.swing.JComboBox
- com.sun.java.swing.JList
- java.awt.Choice
- java.awt.List

d. Enter code in the initialization method to add choices. For a JComboBox, Choice, or List bean, use the *addItem()* method:

```
myChoices.addItem("East");
myChoices.addItem("West");
myChoices.addItem("South");
myChoices.addItem("North");
```

For a JList bean, use the *setListData()* method:

```
String[] data = {"East", "West", "South", "North"};
myChoices.setListData(data);
```

e. Modify the get method for the bean you are initializing, for example, getJComboBox1(). In user code block 1, add code to call the initialization method you just created. The method call should look like this:

```
initializeChoices(instanceName);
```

Specify the instance name for the bean as the *instanceName* argument for the method call. The default instance name is something like *ivjJComboBox1*, *ivjChoice1*, *ivjJList1*, or *ivjList1*.

Defining the selection mode for a JList or List bean

You can define a list to allow either a single selection or multiple selections. With single selection, the previous selection is deselected when the user selects another choice. Multiple selection differs between JList and List beans.

- The JList bean supports two modes of multiple selection. Select one of the following choices for the *selectionMode* property in the JList property sheet:
  - *SINGLE_SELECTION*—allows only one choice to be selected at a time
  - *SINGLE_INTERVAL_SELECTION*—allows a range of choices to be selected
  - *MULTIPLE_INTERVAL_SELECTION*—allows multiple choices to be selected, individually or in ranges
- The List bean supports only one mode of multiple selection. Select one of the following choices for the *multipleMode* property in the List property sheet:
  - *False*—allows only one choice to be selected at a time
  - *True*—allows multiple choices to be individually selected

Allowing text entry in a JComboBox bean

Set the *editable* property to *True* in the property sheet.

Defining the orientation of a JProgressBar, JSlider, JScrollBar, or ScrollBar bean

Select a choice for the *orientation* property in the property sheet. The orientation choices are *VERTICAL* and *HORIZONTAL*.

Defining the value range for a JProgressBar, JSlider, JScrollBar, or ScrollBar bean

Set the *minimum* and *maximum* properties in the property sheet.

Defining the initial value of a JProgressBar, JSlider, JScrollBar, or ScrollBar bean

Set *value* property in the property sheet. The initial value determines the initial progress, selection, or scrolling position in the value range.

Defining tick marks or values for a JSlider bean

To define the value increment between tick marks, set the *majorTickSpacing* and *minorTickSpacing* properties in the JSlider property sheet. To automatically adjust a user selection to the closest tick mark, set the *snapToTicks* property to *True*. To show the tick marks, set the *paintTicks* property to *True*. To show the tick values, set the *paintLabels* property to *True*. To reverse the minimum and maximum ends of the slider, set the *inverted* property to *True*.

Defining scrolling increments for a JScrollBar or ScrollBar bean

To define the value change when the user clicks on a scroll arrow, set the *unitIncrement* property in the ScrollBar property sheet. To define the value change when the user clicks in the scroll bar range away from the scroll box, set the *blockIncrement* property.

Defining tool tip text

For Swing components, you can specify tool tip text, also known as fly-over text or hover help. Enter text for the *toolTipText* property in the component's property sheet.

Defining initial availability

By default, the component is initially enabled for user interaction. To initially disable the component, set the *enabled* property to *False* in the component's property sheet.

3. Provide runtime logic for the component.

Obtaining the selected choice or value

Connect a property representing the selection to a target property. Then, open properties for the connection and select a source event that indicates when the selection changes.

Bean	Source property	Source event
JComboBox	*selectedItem*	*itemStateChanged*
Choice	*selectedItem*	*itemStateChanged*
JList	*selectedValue* or *selectedValues*	*valueChanged*
List	*selectedItem* or *selectedItems*	*itemStateChanged*
JProgressBar	*value*	*stateChanged*
JSlider	*value*	*stateChanged*
JScrollBar	*value*	*adjustmentValueChanged*
ScrollBar	*value*	*adjustmentValueChanged*

To get the value of the selected choice, connect the Choice *selectedItem* property to the value target. To get the index of the selected choice, connect the Choice *selectedIndex* property to the value target.

Getting the value that a user enters in a JComboBox bean

If you set the *editable* property of a JComboBox bean to *True*, the user can enter a value instead of selecting a choice from the list. To get an entered value, do the following:

a. Tear off the *editor* property of the JComboBox bean.

b. Tear off the *editorComponent* property of the torn-off *editor* property.

c. Connect the *keyReleased* event of the torn-off *editorComponent* property to the property that is to receive the entered value.

d. Connect the *value* parameter of the previous connection to the *item* property of the torn-off *editor* property.

Calling a method when the value changes
To call a method when the value changes, connect a source event that indicates when the selection changes to the method.

Obtaining the selected index or indexes for a JComboBox, Choice, JList, or List bean
Connect the *selectedIndex* or *selectedIndexes* property to the value target. Then, open properties for the connection and select a source event that indicates when the selection changes.

Setting the value of a JProgressBar bean
Connect the value to the JProgressBar *value* property. Then, open properties for the connection and select a source event that indicates when the selection changes.

Disabling or enabling a component
Connect a related event to the component's *enabled* property. Specify the parameter value for this connection in one of the following ways:

- To disable the component, open the connection's properties and set the parameter value to *False*.
- To enable the component, open the connection's properties and set the parameter value to *True*.
- To set the new state from another Boolean property, connect the parameter to the other Boolean property.

For examples, see the *BookmarkList* class in the *com.ibm.ivj.examples.vc.swing.bookmarklist* and *com.ibm.ivj.examples.vc.bookmarklist* packages, the *ToDoList* class in the *com.ibm.ivj.examples.vc.todolist* package, and the *Amortization* class in the *com.ibm.ivj.examples.vc.mortgageamortizer* package. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 34. List and Slider Beans" on page 167

BookmarkList Sample

Adding a Button Component

VisualAge provides button beans from the *com.sun.java.swing* and *java.awt* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

You can add a button bean to enable the user to perform an action or select a state.

1. Add one of the following button beans:

Bean	Description
"JButton" on page 175 or "Button" on page 173	A push button, generally used to perform a function

Bean	Description
“JCheckBox” on page 175 or “Checkbox” on page 174	A setting button that is checked when selected
“JRadioButton” on page 176 or “CheckboxGroup” on page 174	A radio button or group for mutually exclusive settings
“JToggleButton” on page 176	A two-state push button that appears to be pushed in when selected

2. Define initial characteristics of the button component.

   Defining a button label

   Enter text for the *text* or *label* property in the component’s property sheet.

   Adding or customizing graphic images for Swing buttons

   You can add images or replace default images. For example, you can add graphics to a JButton or change the default images of a JCheckBox. Select or specify a graphic file for one or more image properties in the component’s property sheet.

   - *icon* represents an unselected button
   - *pressedIcon* represents a button that is pressed
   - *selectedIcon* represents a selected button
   - *disabledIcon* represents an unselected button that is disabled
   - *disabledSelectedIcon* represents a selected button that is disabled
   - *rolloverIcon* represents an unselected rollover button
   - *rolloverSelectedIcon* represents a selected rollover button

   Defining tool tip text for a button component

   For Swing components, you can specify tool tip text, also known as fly-over text or hover help. Enter text for the *toolTipText* property in the component’s property sheet.

   Defining initial availability

   By default, the component is initially enabled for user interaction. To initially disable the component, set the *enabled* property to *False* in the component’s property sheet.

   Defining the initial state of a toggle component

   Set the *selection* or *state* property in the component’s property sheet. For initial selection, set the property value to *True*. Otherwise, the value should be *False*. If the component is one of a group of mutually exclusive choices, only one member of the group should be initially selected.

3. If applicable, assign a toggle component to a group of mutually exclusive choices. Only one member of the group can be selected at a time.

   Adding a group of JToggleButton or JRadioButton beans

   Use a ButtonGroup to define a group of buttons:

   a. Add the buttons for the group.

   b. Add a ButtonGroup bean to the free-form surface.

   c. Save the composite bean.

   d. On the Methods page, add user code in the get method of each button to add the button to the button group. For example, to add JRadioButton1 to ButtonGroup1, add this code to the *getJRadioButton1()* method:

```
getButtonGroup1.add(ivjJRadioButton1);
```

Adding a group of Checkboxes as radio buttons
Use a CheckboxGroup bean to define a group of radio buttons:

a. Add the check boxes for the group.

b. Add a CheckboxGroup bean to the free-form surface.

c. In the property sheet of each Checkbox bean, specify the get method of the CheckboxGroup for the *checkboxGroup* property. For example, to add Checkbox to CheckboxGroup1, specify *getCheckboxGroup1()* for the *checkboxGroup* property.

4. Provide runtime logic for the button component.

Calling a method when a button is selected
Connect the button's *actionPerformed(java.awt.event.ActionEvent)* event to the method on the target bean.

Obtaining the selected choice from a group
From the group's popup menu, tear off the property that represents the selected choice. For a ButtonGroup, tear off the *selection* property. For a CheckboxGroup, tear off the *selectedCheckbox* property. After tearing off the property, you can make connections to features of the Variable that represents the selected choice property.

Disabling a button
Connect a related event to the button's *enabled* property and set the connection parameter value to *False*.

Enabling a button
Connect a related event to the button's *enabled* property and set the connection parameter value to *True*.

For examples, see the *ToDoList* class in the *com.ibm.ivj.examples.vc.todolist* package, and the *JRadioButtonPanel* and *RadioButtonPanel* classes in the *com.ibm.ivj.examples.vc.utilitybeans* package. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Tearing Off Properties" on page 56

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 35. Button Beans" on page 173

Adding a Menu or Tool Bar

VisualAge provides menu beans from the *com.sun.java.swing* and *java.awt* packages, and tool bar beans from the *com.sun.java.swing* and *com.ibm.uvm.abt.edit* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

You can add a menu for a window, for a window component, or for another menu. Define a menu with menu choices that call a method or select a setting.

Define a tool bar with buttons and other components that call a method. Tool bars most commonly contain buttons with graphical images for functions such as clipboard and file operations. A JToolBar can contain other components, however, such as a JComboBox with font choices. The JToolBarButton bean is provided as a convenient means of adding a JButton that is tailored for a tool bar. The JToolBarSeparator bean represents a method call to add separation between other components in a JToolBar.

1. Add one of the following menu or tool bar beans:

Bean	Description
“JMenu” on page 180 or “Menu” on page 185	A cascade menu for another menu
“JMenuBar” on page 181 or “MenuBar” on page 186	A menu bar for a window
“JPopupMenu” on page 182 or “PopupMenu” on page 188	A pop-up menu for window components
“JToolBar” on page 184	A graphical set of tool choices
“JToolBarButton” on page 184	A button for a tool bar
“JToolBarSeparator” on page 185	A visual separator between components in a tool bar

Adding a menu bar to a window

Drop a menu bar bean on the window. The menu bar appears in the window. Additionally, a cascade menu is added on the free-form surface and connected to a new menu label on the menu bar.

Adding a menu to a menu bar

Drop a menu bean on the menu bar. A menu is added on the free-form surface and connected to a new menu label on the menu bar. If you have already dropped the menu bean on the free-form surface, drag it to the menu bar and drop it to make the connection.

Adding a cascade menu to another menu

Drop a menu bean on the base menu you want to cascade from. A menu is added on the free-form surface and connected to a new menu item on the base menu. If you have already dropped the new menu bean on the free-form surface, drag it to the base menu and drop it to make the connection.

Adding a pop-up menu

Drop a pop-up menu bean on the free-form surface. To show the menu when the user clicks the pop-up mouse button on a window component, connect the mouse event for the composite bean to a code that displays the menu.

The following example code determines whether the pop-up mouse button has been clicked, gets the window component, and shows the pop-up menu:

```
protected void genericPopupDisplay(java.awt.event.MouseEvent e, java.awt.PopupMenu p) {
   if ((e.isPopupTrigger()))  {
  e.getComponent().add(p);
  p.show(e.getComponent(), e.getX(), e.getY());
   };
}
```

Adding a tool bar

You can add a JToolBar bean to the content pane of a Swing applet or window. Before you add the tool bar, you should define the *layout* property of the content pane as a border layout. Add the tool bar to one of the border regions (North, South, East, or West). By default, a tool bar is detachable because the *floatable* property is set to *True*. If you want a floatable tool bar to be attachable to any side of the content pane, do not add any other components to the four border regions.

When you add a JToolBar, a JToolBarButton is added with it. The JToolBarButton is actually a JButton that is tailored for a tool bar. Change properties of the JToolBarButton , such as the icon, to serve your purpose.

Adding an action to a menu or tool bar

You can define a subclass of AbstractAction that can be used in one or more Swing menus and tool bars. For example, you can define an action for a clipboard operation and add it to menus and tool bars. You must implement the *actionPerformed()* method of the subclass. Add the AbstractAction subclass to the free-form surface. Save the bean. Then, edit the get method for the menu or tool bar to add the AbstractAction subclass.

2. Enable or disable the menu. When a menu is disabled, the user cannot open it. By default, a menu is initially enabled.

Initially disabling a menu

Set the *enabled* property to *False* in the menu property sheet.

Disabling a menu when an event occurs

Connect the event to the menu’s *enabled* property. Then, set the connection parameter value to *False*.

Enabling a menu when an event occurs

Connect the event to the menu’s *enabled* property. Then, set the connection parameter value to *True*.

3. Add choices to the menu. Select from the following menu choice beans:

Bean	Description
“JCheckBoxMenuItem” on page 180 or “CheckboxMenuItem” on page 179	A menu choice that toggles a setting on and off
“JMenuItem” on page 182 or “MenuItem” on page 187	A menu choice that calls a method
“JRadioButtonMenuItem” on page 183	A menu choice that provides one of a set of mutually exclusive setting values
“JSeparator” on page 183 or “MenuSeparator” on page 187	A horizontal line that separates groups of related choices

Adding a choice to a menu

Drop one of the menu choice beans within a menu at the location you want. Before you release the mouse button, a horizontal cursor line indicates where the choice will be placed.

Moving a menu choice

Drag the menu choice and drop it at a new location, either in the same menu or in another menu. Before you release the mouse button, a horizontal cursor line indicates where the choice will be placed.

Adding a separator to a menu

Drop a separator bean within a menu at the location you want. Before you release the mouse button, a horizontal cursor line indicates where the separator will be placed.

Adding a component to a tool bar
Drop a component bean on a JToolBar at the location you want. Before you release the mouse button, a cursor line indicates where the choice will be placed.

Moving a component on a tool bar
Drag the component and drop it at a new location. Before you release the mouse button, a cursor line indicates where the choice will be placed.

Adding a separator to a tool bar
Drop a JToolBarSeparator bean on a JToolBar at the location you want. Before you release the mouse button, a cursor line indicates where the separator will be placed.

4. Define each menu choice to indicate its purpose and, optionally, to provide a keyboard shortcut.

Defining text for a menu choice
Enter the text in the value field of the *label* property in the property sheet.

Defining a keyboard shortcut for a menu choice
Set the *shortcut* property in the property sheet for the menu choice. Select the **Shift** check box if you want to use the Shift key. Select a unique key choice for the menu item.

5. Define the function or setting for each menu choice.

Calling a method from a menu choice
Connect the *actionPerformed(java.awt.event.ActionEvent)* event of the menu choice to the method on the target bean. The method is called when the user selects the menu choice.

Defining the initial state of a check box menu choice
Set the *state* property of the menu choice in its property sheet. If the property value is *True*, the setting is initially on. Otherwise, the setting is initially off.

Note: This property setting does not affect the appearance of a check box menu item in the Visual Composition Editor.

Updating a setting for a check box menu choice
Connect the *state* property of the menu choice to the property on the target bean. The property is set when the user toggles the menu choice.

6. Enable or disable menu choices as appropriate. For example, a menu choice might have no meaning when another menu choice is toggled on, or when the user has provide no input in a text field. When a menu choice is disabled, the user cannot select it. By default, a menu choice is initially enabled.

Initially disabling a menu choice
Set the *enabled* property to *False* in the menu choice property sheet.

Disabling a menu choice when an event occurs
Connect the event to the menu choice's *enabled* property. Then, set the connection parameter value to *False*.

Enabling a menu choice when an event occurs
Connect the event to the menu choice's *enabled* property. Then, set the connection parameter value to *True*.

For examples, see the *PopupMenuExample* class in the *com.ibm.ivj.examples.vc.popupmenuexample* package, the *Amortization* class in the *com.ibm.ivj.examples.vc.mortgageamortizer* package, and the *DirectoryExplorer* class in the *com.ibm.ivj.examples.vc.swing.directoryexplorer* package. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 22. Using VisualAge Beans in Visual Composition” on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

Dynamically Creating and Accessing a Bean Instance

VisualAge provides beans that enable you to dynamically create and reference bean instances visually. A Factory creates new instances of a bean type, based on a connection from an event to a constructor for the Factory’s type. A Variable references any instance of the Variable’s type that you assign to it using a connection. With either a Factory or a Variable, you specify the bean type that it can create or reference.

A Factory’s type specifies the type of bean instance, or object, that it creates. A Variable’s type specifies the type of object that can be assigned to it. For example, if you change a Factory’s type to Customer, it can create Customer objects. If you change a Variable’s type to Customer, you can use it to reference any Customer object that you assign to it.

You can visually create and access beans in the Visual Composition Editor.

1. Add one of the following beans and specify the type of bean it represents:

Bean	Description
“Factory” on page 189	A bean that dynamically creates instances of Java beans
“Variable” on page 190	A bean that provides access to instances of Java beans

Adding a Factory or Variable bean

a. Select a Factory or Variable bean from the **Other** category of the beans palette. Alternatively, you can select a class type as a Variable in the Choose Bean window.

b. Add the bean to the free-form surface of the composite bean you are composing.

Changing the Factory or Variable type

When you add a Factory or Variable bean from the palette, its initial type is Object. Change the type as follows:

a. Open the pop-up menu of the bean.

b. Select **Change type** to open the Choose a Type window.

c. Enter the full or partial type name in the **Pattern** field.

d. Select the type you want in the **Class Names** or **Type Names** field.

e. If the type is found in more than one package, select the package in the **Package Names** field that contains the one you want.

f. Select **OK**.

After you change the type, you can make connections to features of the new type.

Changing the bean name

You might want to change the name of a Factory or Variable bean to a more descriptive name for visual composition. This can be particularly helpful for a Variable in some circumstances:

- If you change the Variable type. The initial name of the Variable reflects its initial type.
- If you promote a feature from a Variable to a composite bean. The Variable name is reflected in the default name of the promoted feature.

Change the name as follows:

a. Open the pop-up menu of the Factory or Variable bean.

b. Select **Change Bean Name** to open the Bean Name Change Request window.

c. Enter the new name.

d. Select **OK**.

2. Define when and how how the Factory or Variable bean is used.

Creating objects with a Factory

Connect an event to a Factory constructor method. If the constructor you choose requires parameter values, provide these values either with additional connections or property settings. Because the Factory references an object that it creates until it creates another object, you can make connections from its *this* event to methods and properties of the object it references.

Assigning a bean instance to a Variable

Connect a bean property of the same type as the Variable to the *this* property of the Variable. The connection assigns the source property to the Variable, so the Variable references the source as a bean instance. If the source bean itself is the source property , use its *this* property as the connection source.

Two customized variations of this procedure are commonly used:

Tearing off a property

You can gain access to features of a bean that is a property of another bean by detaching a reference to the property as a bean instance in a Variable. Then, you can access the features of the property through the Variable. This procedure is called tearing off a property.

For example, you might use a Customer bean that has *name*, *address*, and *phone* properties. The *address* property is an Address bean that has *street*, *city*, *state*, and *zipCode* properties. When you add a Customer bean, you can make connections to its *address* property, but not to individual elements of the address. If you tear off the *address* property of the Customer bean, an Address Variable is placed on the free-form surface. A connection assigns the *address* property of the Customer bean to the Address Variable. You can make connections to the properties of the Address Variable to access elements of the Customer's *address* property.

Promoting a Variable

You can enhance bean reusability by defining its data source as a property of the bean. When you use the bean in another bean, you can assign the data using a connection to the data property. To do this, add a Variable for the data source bean type in the reusable bean. Then, promote the Variable's *this* property to the interface of the reusable bean.

For example, you might compose a CustomerView bean that provides a panel of fields to display or obtain information for a Customer bean. To make the CustomerView bean reusable wherever you might use a Customer bean, you don't want to specify a particular Customer bean as the data model for the CustomerView bean. You can accomplish this as follows:

a. Use a Customer Variable bean, instead of a Customer bean, in the CustomerView bean as the data model for the customer information fields.

b. Connect properties of the Customer Variable to corresponding customer information fields to tie the data model to the user interface.

c. Promote the Customer Variable to the CustomerView bean interface as a *customer* property.

d. Whenever you add a CustomerView bean to another bean, also add either a Customer bean or another bean that contains a Customer bean as a property. Connect the Customer bean to the *customer* property of the CustomerView bean. This assigns the Customer bean to the Customer Variable in the CustomerView bean.

For examples that use a Variable, see the *Amortization* class in the *com.ibm.ivj.examples.vc.swing.mortgageamortizer* package, and the *AddressView* and *CustomerView* classes in the *com.ibm.ivj.examples.vc.customerinfo* package. These examples are shipped in the IBM Java Examples project.

RELATED TASKS

"Chapter 16. Working with Beans Visually" on page 53

"Chapter 17. Composing Beans Visually" on page 67

"Promotion of Bean Features" on page 23

"Tearing Off Properties" on page 56

"Chapter 22. Using VisualAge Beans in Visual Composition" on page 89

Adding the IBM Java Examples project

RELATED REFERENCES

"Chapter 37. Factory and Variable Beans" on page 189

CustomerInfo Sample

Chapter 23. Enabling Custom Edit Support for Your Bean

The JavaBeans specification defines two ways for you to implement custom edit behavior for your bean: property editors and customizers. Check the most recent version of the spec for details; a summary of custom editors follows:

- For setting a single property, use a property editor. This simple GUI implements the *java.beans.PropertyEditor* interface. You can associate a property editor with a property type or a specific property. Property editors are typically grouped into a single *property sheet* for the bean. The quickest way to create a property editor is to inherit from *java.beans.PropertyEditorSupport*, a concrete implementation class.
- For setting multiple properties through a single GUI, use a customizer. It is also a good choice if the bean interface itself is large or complex enough that you want to take responsibility for edit behavior for all properties in the bean. The quickest way to create a customizer in VisualAge is to inherit from *java.awt.Panel*, implementing the *java.beans.Customizer* interface.

All properties must be serializable. Java uses serialization to share instance information.

You must explicitly assign a customizer for it to be used; this is not true for property editors. The JavaBeans specification provides the following alternatives for associating a property editor and property:

- Name the property editor class appropriately and place it in the same package. If the property is of type MyObject, call the property editor class MyObjectEditor.
- Register the property editor with the PropertyEditorManager.
- Explicitly assign the property editor in the bean's BeanInfo class.

In VisualAge, you can assign property editors and customizers from the BeanInfo page of the class browser, or you can hand-edit the BeanInfo class directly.

To assign a property editor when you define the property, enter the name of the class in the **Property editor** field on the second page of the New Property Feature SmartGuide. To assign a property editor at any other time, directly edit the **Property editor** field in the **Property Feature Information** pane of the BeanInfo page. In this case, the change does not take effect until you save the bean, either by closing the class browser and electing to save changes or by typing Ctrl+S.

To assign a customizer to a bean, directly edit the **Customizer class** field in the**Bean Information** pane of the BeanInfo page. (Make sure you have no features selected; if a feature is selected, the Feature Information pane appears instead.) The change does not take effect until you save the bean, either by closing the class browser and electing to save changes or by typing Ctrl+S.

To test a bean's customization, drop the bean on the free-form surface and double-click on it.

- If you have implemented a property editor, the standard property sheet appears. Select the value field for the associated property. If the property editor has a custom editor, a small button ... appears to the right. Select the button to open the custom editor.

- If you have implemented a customizer, a property sheet appears, bearing a **Custom Properties** button. Select the button to open the customizer; it appears in a modal window.

For more information on implementing property editors themselves, see "Chapter 24. Property Editor Examples" on page 119, which discusses the *com.ibm.ivj.examples.vc.propertyeditors* package.

RELATED CONCEPTS

"Chapter 13. Object Serialization in VisualAge" on page 47

RELATED TASKS

"Adding Property Features" on page 133

RELATED REFERENCES

"Chapter 56. BeanInfo Page" on page 237

Chapter 24. Property Editor Examples

The JavaBeans specification and VisualAge support the following types of editors implementing the *java.beans.PropertyEditor* interface:

Tag-based. This editor presents a fixed list of property values (known individually as *tags*) from a drop-down list in the property sheet.

Text-based. This editor accepts a single string from within a property sheet, parsing it as necessary to set the property.

Custom. This editor opens a window separate from the property sheet to collect settings information.

Paintable. This editor paints a graphic representation of the property value back into the property sheet rather than returning a String value.

Each type of property editor supports a subset of the PropertyEditor interface, meaning that a different set of method implementations returns non-null values. A summary follows:

- All property editors must support setValue( ) and signal property change events. They must also support either setAsText( ) or getCustomEditor( ). If they support getCustomEditor( ), they must also return `true` from supportsCustomEditor( ).
- Tag-based editors support getTags( ) and getAsText( ).
- Text-based editors support getAsText( ).
- Paintable editors support paintValue( ) and return `true` from isPaintable( ).

For properties that require special code to be generated for initialization, the getJavaInitializationString( ) method should return a non-null value. VisualAge uses this value when it generates code for the bean whose property you are setting. (For an example, see “Custom Editor for the Person Bean” on page 122.)

For examples, look at the *com.ibm.ivj.examples.vc.propertyeditors* package shipped with VisualAge in the IBM Java Examples project. Consider the Person bean, which has the following properties: *name*, *address*, *phoneNumber*, *sex*, and *incomeRange*.

```
public class Person {
   String fieldSex = "";
   protected transient java.beans.PropertyChangeSupport propertyChange
      = new java.beans.PropertyChangeSupport(this);
   transient Address fieldAddress = null;
   static public final int belowTwenty = 1;
   static public final int twentyToFifty = 2;
   static public final int fiftyToOneHundred = 3;
   static public final int aboveOneHundred = 4;
   int fieldIncomeRange = 0;
   String fieldPhoneNumber = "";
   Name fieldName = null;
}
```

Because possible *incomeRange* values are predefinable, this property can be set using a tag-based editor. The *phoneNumber* property has a known format in each locale, so the example uses a text-based editor and validates the input argument in setAsText( ) to make sure it matches the North American notion of a telephone number. For the *name* property, the example uses a custom editor instead of setAsText( ); *name* is an instance of a serializable class, as required. We illustrate a paintable editor for the *sex* property. Because this example does not include an editor for the *address* property and the Address class happens not to be serializable, the instance has been marked as transient.

The sample package illustrates two means of associating a property editor. The *sex*, *incomeRange*, and *phoneNumber* properties are of types (String and int) for which there is default editor support in VisualAge, so editors for these properties are explicitly assigned in the bean's BeanInfo class. The *name* property is of type Name; because a NameEditor class exists in the same package, Java uses it by default.

For a tour of each property editor, follow the Related Reference links below. To test these editors, we used two simple visual composites based on Applet, PersonTester and PaintTester. When you double-click on the Person1 bean in each composite, a standard property sheet appears with certain property editor examples enabled.

RELATED TASKS

"Chapter 23. Enabling Custom Edit Support for Your Bean" on page 117

RELATED REFERENCES

"Tag-Based Editor for the Person Bean"

"Text-Based Editor for the Person Bean" on page 121

"Custom Editor for the Person Bean" on page 122

"Paintable Editor for the Person Bean" on page 125

Tag-Based Editor for the Person Bean

This topic discusses classes found in the *com.ibm.ivj.examples.vc.propertyeditors* package. You can try this editor from either PersonTester or PaintTester.

The *incomeRange* property has a tag-based editor class associated with it. This editor extends *java.beans.PropertyEditorSupport*, a concrete implementation class for the *java.beans.PropertyEditor* interface. As a result, *setValue( )* does not have to be implemented locally.

```
public class IncomeRangeEditor extends java.beans.PropertyEditorSupport {
   String[] stringValues = {
      "0 - 20,000",
      "20,000 - 50,000",
      "50,000 - 100,000",
      "100,000+" };
   int[] intValues = {1, 2, 3, 4};
   String[] codeGenStrings = {
      "propertyeditors.Person.belowTwenty",
      "propertyeditors.Person.twentyToFifty",
      "propertyeditors.Person.fiftyToOneHundred",
      "propertyeditors.Person.aboveOneHundred"};

   public String getAsText() {
      for (int i=0; i< intValues.length; i++) {
         if (intValues[i] == ((Integer) getValue()).intValue())
            return stringValues[i];
      }
      return "";
   }
   public String getJavaInitializationString() {
      for (int i=0; i< intValues.length; i++) {
         if (intValues[i] == ((Integer) getValue()).intValue())
            return codeGenStrings[i];
      }
      return "0";
   }
   public String[] getTags() {
      return stringValues;
   }
   public void setAsText(String text) throws java.lang.IllegalArgumentException {
      for (int i=0; i< stringValues.length; i++) {
         if (stringValues[i].equals(text)) {
            setValue(new Integer(intValues[i]));
            return;
         }
      }
      throw new java.lang.IllegalArgumentException(text);
   }
}
```

The *getTags( )* method holds allowable property values in an array. IncomeRangeEditor uses a set of parallel arrays to manage the allowable income ranges. Note that the sole purpose of the codeGenStrings array is to address incomeRange constants in the Person bean through the *getJavaInitializationString( )* method. The return value from this method appears in the *getPerson1( )* method of the PersonTester class:

```
private Person getPerson1() {
if (ivjPerson1 == null) {
try {
ivjPerson1 = new PropertyEditors.Person();
ivjPerson1.setSex("Female");
ivjPerson1.setName(new PropertyEditors.Name("Mrs.", "Susan", "Gail", "Carpenter"));
ivjPerson1.setPhoneNumber("555-1212");
ivjPerson1.setIncomeRange(propertyeditors.Person.belowTwenty);
// user code begin {1}
// user code end
} catch (java.lang.Throwable ivjExc) {
// user code begin {2}
// user code end
handleException(ivjExc);
}
};
return ivjPerson1;
}
```

RELATED REFERENCES

"Chapter 24. Property Editor Examples" on page 119

"Text-Based Editor for the Person Bean"

"Custom Editor for the Person Bean" on page 122

"Paintable Editor for the Person Bean" on page 125

Text-Based Editor for the Person Bean

This topic discusses classes found in the *com.ibm.ivj.examples.vc.propertyeditors* package.You can try this editor from either PersonTester or PaintTester.

The *phoneNumber* property has a text-based editor associated with it. This editor extends *java.beans.PropertyEditorSupport*, a concrete implementation class for the *java.beans.PropertyEditor* interface. As a result, setValue( ) does not have to be implemented locally.

```
public class PhoneNumberPropertyEditor extends java.beans.PropertyEditorSupport {
   public void setAsText(String text) throws java.lang.IllegalArgumentException {
      if ((text.length() == 8) && (text.charAt(3) == '-' )) {
         setValue(text);
```

```
            return;
         }
         if (text.length() == 7) {
            setValue(text.substring(0,3) + "-" + text.substring(3,7));
            return;
         }
         throw new java.lang.IllegalArgumentException(text);
      }
   }
```

The *setAsText( )* method accepts only values that meet its format criteria; otherwise, it throws an IllegalArgumentException.

RELATED REFERENCES

"Chapter 24. Property Editor Examples" on page 119

"Tag-Based Editor for the Person Bean" on page 120

"Custom Editor for the Person Bean"

"Paintable Editor for the Person Bean" on page 125

Custom Editor for the Person Bean

This topic discusses classes found in the *com.ibm.ivj.examples.vc.propertyeditors* package.You can try this editor from PersonTester only.

The *name* property has a custom editor (NameEditor) and custom editor panel (NameCustomEditor) associated with it. Support of the PropertyEditor interface and the edit function itself are decoupled in order to optimize performance in Person's property sheet. By decoupling them, we can delay construction of the custom editor panel until the user requests it; only the property editor itself is instantiated when the property sheet is first opened. This functional separation is significant when the type being supported can occur several times in a single property sheet (like a custom String editor), because each property requires its own instance of the custom editor.

When you browse the sample, note the public constructor for NameEditor. All of the other property editor classes in this sample have protected constructors, generated by default because the superclass constructor is protected. When you explicitly assign a property editor in BeanInfo, access to the protected constructor is not a problem. For the *name* property, however, we have not explicitly assigned an editor in BeanInfo. In this case, the PropertyEditorManager class becomes involved in coordinating edit support for the property, so we must provide a public constructor.

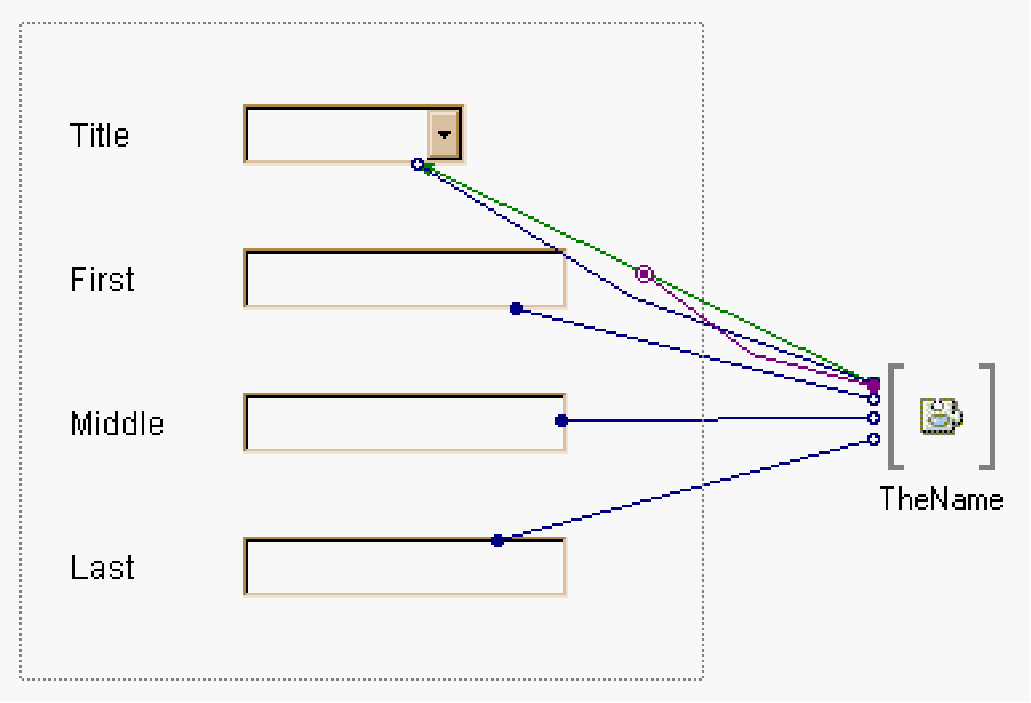
The NameEditor property editor looks like this:

```
public class NameEditor extends java.beans.PropertyEditorSupport {
   java.beans.PropertyChangeSupport iPropertyChange
      = new java.beans.PropertyChangeSupport(this);
   NameCustomEditor iNameCustomEditor = null;
   Name iName = null;

   public String getAsText() {
      return ((Name) getValue()).toString();
   }
   public java.awt.Component getCustomEditor() {
      if (iNameCustomEditor == null) {
         iNameCustomEditor = new NameCustomEditor();
         iNameCustomEditor.setTheNameThis( getName() );
      }
      return iNameCustomEditor;
   }
   public String getJavaInitializationString() {
      Name tName = ( (Name) getValue() );
      return "new propertypditors.Name(\"" +
         tName.getTitle() +
         "\", \"" +
         tName.getFirstName() +
         "\", \"" +
         tName.getMiddleName() +
         "\", \"" +
         tName.getLastName() +
         "\")";
   }
   public Name getName() {
      if (iName == null) iName = new Name();
      return iName;
    }

   public Object getValue() {
      if (iNameCustomEditor == null)
         return getName();
      else
         return iNameCustomEditor.getTheNameThis();
   }
   public void setValue(Object value) {
      Object tValue = getName();
      if (iNameCustomEditor == null) {
         iName = ((Name) value);
         iPropertyChange.firePropertyChange("value", tValue, value);
      }
      else
      iNameCustomEditor.setTheNameThis( (Name) value );
}
   public boolean supportsCustomEditor() {
      return true;
   }
}
```

NameEditor manages the getting and setting of property values through the property sheet; NameCustomEditor collects the information from the user. The common currency between the property editor and the custom editor panel is a Name instance. The NameCustomEditor panel looks like this:



In this composite, property-to-property connections link Name properties in the variable bean to the *text* properties of the TextField beans. The Choice bean requires two connections: one from its *selectedItem* property to the *title* property of the variable, and one from the *title* property of the variable to the select( ) method of the Choice bean (passing in the current value of *title* as a parameter of the select( ) method). The variable's *this* property is promoted to the interface of the composite so that it can be set from NameEditor during initialization.

To understand how these two classes interact, follow this partial program flow:

1. When Person's property sheet is opened, a NameEditor instance is created. The *getAsText( )* method is called to populate the value field for the *name* property.
2. The *setValue( )* method is called, passing the value that currently appears in the property sheet as an input parameter. This value is then stored in NameEditor's iName field.
3. When the value field is selected, NameEditor's *supportsCustomEditor( )* method is called. It returns `true`, so a small button appears in the value field.
4. When the button is selected, NameEditor's *getCustomEditor( )* method is called, creating an instance of NameCustomEditor and setting NameCustomEditor's TheName variable to match NameEditor's iName field.
5. If the **OK** button of NameCustomEditor is selected, the *getValue( )* and *getJavaInitializationString( )* methods of NameEditor are called. If, however, the **Cancel** button is selected, no methods are called in NameEditor.

RELATED REFERENCES

"Chapter 24. Property Editor Examples" on page 119

"Tag-Based Editor for the Person Bean" on page 120

"Text-Based Editor for the Person Bean" on page 121

“Paintable Editor for the Person Bean”

Paintable Editor for the Person Bean

This topic discusses classes found in the *com.ibm.ivj.examples.vc.propertyeditors* package.You can try this editor from PaintTester only.

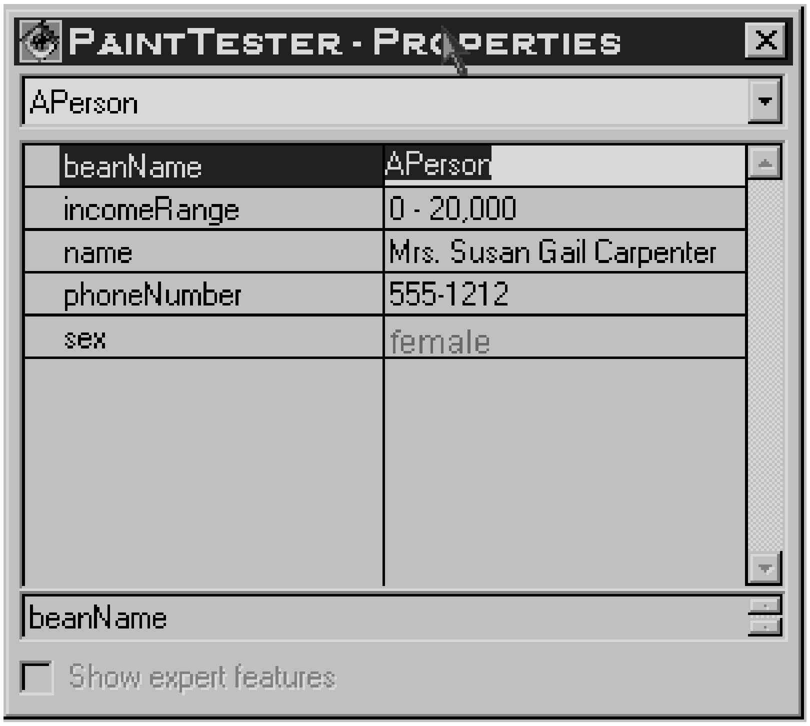
The *sex* property has a paintable editor associated with it. This editor extends *java.beans.PropertyEditorSupport*, a concrete implementation class for the *java.beans.PropertyEditor* interface. As a result, setValue( ) does not have to be implemented locally. In this type of property editor, we paint the tagged value back into the property sheet instead of returning it as a String.

```
public class SexEditor extends java.beans.PropertyEditorSupport {
   public String[] getTags() {
      String[] tags = {"male", "female"};
      return tags;
   }
   public boolean isPaintable() {
      return true;
   }
   public void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box) {
      String tString = getAsText();

      if (tString.equals("male") )
         gfx.setColor(java.awt.Color.blue);
      else
         gfx.setColor(java.awt.Color.magenta);

      gfx.drawString(tString, (box.x) + 1, (box.y) + (box.height) - 2);
      return;
   }
}
```

The *getTags( )* method holds allowable property values in an array. Instead of using *getAsText( )* and *setAsText( )* as we did for IncomeRangeEditor, we override *isPaintable( )* and *paintValue( )*. The resulting property sheet looks like this:



APerson

beanName	APerson
incomeRange	0 - 20,000
name	Mrs. Susan Gail Carpenter
phoneNumber	555-1212
sex	female

beanName

□ Show expert features

RELATED REFERENCES

"Chapter 24. Property Editor Examples" on page 119

"Tag-Based Editor for the Person Bean" on page 120

"Text-Based Editor for the Person Bean" on page 121

"Custom Editor for the Person Bean" on page 122

Chapter 25. Separating Strings for Translation

Before doing this task, please read the conceptual information listed at the end of this topic.

From the Workbench, you can separate all String values from the class at once. From the Visual Composition Editor, you can separate String property values as you set them in the property sheet.

To separate String values from an entire class at once, follow these steps:

1. From the Projects page of the Workbench, select the class.
2. Select **Selected** and then **Externalize Strings**. Alternatively, click mouse button 2 and select **Externalize Strings** from the pop-up menu that appears.

   The Externalizing: Package.Class window appears, bearing a list of hardcoded strings found in the class.
3. Specify the type of resource bundle by selecting one of the following radio buttons:
   - **List resource bundle**
   - **Property resource file**
4. Specify the name of the resource bundle.
   - To choose an existing resource bundle, select the **Browse** button, pick a bundle from the standard dialog box, and select **OK**.
   - To create a new bundle, select **New**. Enter values as prompted, depending on the type of resource bundle; select **OK**.
5. If necessary, mark for exclusion those strings listed under **Strings to be separated** that should be left as is. To mark an item, select the graphic listed to the left of the column, as follows:
   - To separate the item, do nothing. ☑ **Translate** is already displayed.
   - If the item must never be separated, select ☑ once to display ☒ **Never translate**.
   - To leave the item hardcoded for now, select ☑ twice to display ? **Skip**.

   If you are not sure of an item, review it in the **Context** field.
6. Select **OK** to proceed with separation.

VisualAge marks each item marked ☒ with a special comment. To make a string previously marked ☒ appear in the externalization list once again, find the string in the code and delete the comment at the end of the line: `//$NON-NLS-1$`. Then perform this task a second time.

Separating Strings through Property Sheets

To separate String property values as you set them, follow these steps:

1. Open the property sheet for each embedded bean that contains a text setting.
2. Select the value field to the right of the property name. A small button ... appears to the right.

3. Select the small button. The String Externalization Editor window appears. At the top of the window is a set of radio buttons that enable you to specify how you want VisualAge to handle the text. The current property setting, if any, appears in the **Value** field.
4. Select the appropriate radio button:
   - **Do not externalize string**
   - **Externalize string**
5. If you selected **Do not externalize string**, you are finished. Just select **OK** to close the window.
6. If you selected **Externalize string**, specify the type of resource bundle by selecting one of the following radio buttons:
   - **List resource bundle**
   - **Property resource file**
7. Specify the name of the resource bundle.
   - To choose an existing resource bundle, select the **Browse** button, pick a bundle from the standard dialog box, and select **OK**.
   - To create a new bundle, select **New**. Enter values as prompted, depending on the type of resource bundle; select **OK**.

   The name of the bundle appears in the **Bundle** list. If you selected an existing class, a currently defined key-value pair appears in fields below the bundle name.
8. To define a resource, type its name in the **Key** field. If the resource already exists, the corresponding value for the key appears in the **Value** field underneath; otherwise, the field is empty. To edit the resource, type a new value.
9. Select **OK** to close the window.

The next time you save the class, VisualAge modifies the generated get methods for the beans whose properties you just set as bundles.

RELATED CONCEPTS

"Chapter 14. Internationalization in VisualAge" on page 49

RELATED TASKS

"Editing Bean Properties" on page 56

RELATED REFERENCES

"Chapter 54. String Externalization Editor" on page 233

"Chapter 55. Externalizing: Package.Class" on page 235

"Chapter 11. Example of Code Generated from Visual Composite" on page 41

Chapter 26. Incorporating User-Written Code into Visual Composites

Although VisualAge enables you to compose and generate user interface beans, you will probably want to write other beans yourself at some point. These are typically nonvisual beans that provide business logic. You can either create a new bean and write the code to support its features, or you can define bean interface features for code you have already written.

If you just need to extend the function of the bean, you can probably accomplish this by using code connections. As a last resort, you can modify generated code for the bean.

Assembling a Bean from Generated and User-Written Code

Before you get started, read the related conceptual topic about generated code.

1. Design the bean interface.
2. Define the bean interface in the BeanInfo page. VisualAge generates source code for the interface features in the bean class. It also generates descriptor methods for the bean and its features in an associated BeanInfo class.
3. Modify the feature code to provide the behavior you want.

Modifying Generated Feature Code

For properties, generated feature code is usually sufficient without modification. For method features, you must modify the feature code to add the behavior you want your bean to provide.

You can modify the feature code in the **Source** pane of either the Methods page or the BeanInfo page. If you choose to modify feature code for visual composites, be sure to stick to the designated user-code areas marked. Otherwise, VisualAge will overwrite your code the next time the bean is saved.

If you need to modify the signature for a method that supports a feature, follow these steps:

1. Remove the feature in the the BeanInfo page.
2. Modify or replace the method in the Methods page.
3. Add the feature again in the the BeanInfo page.

Adapting User-Written Classes for Use as Beans

Before you get started, read the related conceptual topic about generated code.

1. If you wrote a class outside of VisualAge, import the class. VisualAge interprets the bean interface using introspector design patterns.
2. From the BeanInfo page, extend the bean interface with new features as needed. VisualAge generates a BeanInfo class when you add the first new feature. The BeanInfo class contains bean information for the bean, for the features you imported and for each new feature you add.

If you add public methods on the Methods page, you can add them as features on the BeanInfo page. To add methods as features, select **Add Available Features** from the **Features** menu. Then, select the methods you want to add as features.

If you add a new method feature with the same name as a method you have already written, VisualAge uses the existing method. Otherwise, it generates a new method stub.

RELATED CONCEPTS

“Chapter 3. Visual, Nonvisual, and Composite Beans” on page 5

“Code Connections” on page 29

“Chapter 9. Generated Code” on page 33

RELATED TASKS

“Chapter 27. Defining Bean Interfaces for Visual Composition” on page 131

“How Generated Code Coexists with User-Written Code” on page 37

Importing Classes from the File System

RELATED REFERENCES

“Generated BeanInfo Descriptor Code (an advanced topic)” on page 36

Chapter 27. Defining Bean Interfaces for Visual Composition

The bean interface defines the property, event, and method features of your bean. These features can be used in visual composition when your bean is added to another bean. A BeanInfo class describes the bean and features that you add to the bean. Other features are inherited from the superclass of your bean unless you choose not to inherit features. See the related conceptual topic about bean interfaces for more information.

Add features to the bean interface in the BeanInfo page. You can use either the tool bar or **Features** menu to add a new feature. When you add a feature, VisualAge generates the following:

- Public methods for the feature in the bean class
- Bean information code that describes the feature in the BeanInfo class for the bean

If you create public methods for the bean in the Methods page, you can add them as features in the BeanInfo page. Select **Add Available Features** from the **Features** menu to open the Add Available Features window and add methods as features.

Promote features of embedded beans in the Visual Composition Editor. You can promote features from the pop-up menu of an embedded bean. When you promote a feature, VisualAge generates the following:

- Public methods in the bean class that call methods of the embedded bean
- Bean information code that describes the feature in the BeanInfo class for the bean

RELATED CONCEPTS

"Chapter 7. Bean Interfaces and BeanInfo" on page 23

"Chapter 2. How Classes and Beans Are Related" on page 3

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Creating and Modifying a BeanInfo Class" on page 132

"Adding Property Features" on page 133

"Adding Method Features" on page 134

"Adding Event Features" on page 135

"Promoting Features of Embedded Beans" on page 137

"Specifying Expert Features" on page 138

"Specifying Hidden Features" on page 138

RELATED REFERENCES

"Chapter 56. BeanInfo Page" on page 237

"Features—BeanInfo Page" on page 241

Creating and Modifying a BeanInfo Class

When you create a new bean, it does not initially have a BeanInfo class. VisualAge automatically creates a BeanInfo class for a bean if one does not exist and you do any of the following:

- Modify a BeanInfo property in the **Information** pane of the BeanInfo page. VisualAge generates bean information code that describes the bean.
- Add a new feature in the BeanInfo page. VisualAge generates bean information code that describes the bean and the new feature.
- Promote a feature of an embedded bean in the Visual Composition Editor. VisualAge generates bean information code that describes the bean and the promoted feature.

You can explicitly create a BeanInfo class in the BeanInfo page as follows:

1. From the **Features** menu, select **New BeanInfo Class** to open the SmartGuide – BeanInfo Class window.
2. In the SmartGuide – BeanInfo Class window, you can specify a display name and short description to use for the bean. If you want to provide customized initialization of bean properties, specify a customizer class for the bean. Select **Next** to open the SmartGuide – Bean Icon Information window.
3. In the SmartGuide – Bean Icon Information window, you can specify files containing icons for the bean. Select **Finish** to create the BeanInfo class.

You can also create or replace a BeanInfo class in the BeanInfo page as follows. From the **Features** menu, select **Generate BeanInfo class**. VisualAge generates bean information code that describes the bean and all features that you have added or promoted to the bean interface.

To modify the information in a BeanInfo class, edit bean information properties in the **Information** pane of the BeanInfo page. If no feature is selected in the **Features** pane, you can edit bean information for the bean. If a feature is selected, you can edit bean information for the feature.

If you want a bean to be serialized, set the *Hidden-state* property of the bean to *true* in the **Information** pane.

RELATED CONCEPTS

"Chapter 7. Bean Interfaces and BeanInfo" on page 23

"Generated BeanInfo Descriptor Code (an advanced topic)" on page 36

"Chapter 13. Object Serialization in VisualAge" on page 47

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

RELATED REFERENCES

"BeanInfo Class SmartGuide" on page 248

"Bean Icon Information SmartGuide" on page 249

"Information Pane—BeanInfo Page" on page 240

Adding Property Features

Define property features to represent bean data or attributes that you want other beans to have access to.

Add a new property feature in the BeanInfo page as follows:

1. From the tool bar, select **New Property Feature**. If you prefer, you can select **New Property Feature** from the **Features** menu. Either selection opens the SmartGuide – New Property Feature window.
2. In the SmartGuide – New Property Feature window, do the following:
   a. Specify the property name in the **Property name** field.
   b. Specify the property type in the **Property type** field.
   c. If you want the property value to be retrievable, make sure that the **Readable** check box is selected. If this option is selected, a get method is generated for the property.
   d. If you want the property value to be modifiable, make sure that the **Writeable** check box is selected. If this option is selected, a set method is generated for the property.
   e. If you want the property to send value changes on connections, make sure that the **bound** check box is selected.
   f. If the property consists of an array of elements, select the **Indexed** check box. After you finish adding the property feature, select **Add Available Features** from the **Features** menu to add the get and set array element methods as features so you can make connections to them.
   g. If you want other beans to be able to veto value changes for the property, select the **constrained** check box.
   h. Select **Next** to open the SmartGuide – Bean Information window.
3. In the SmartGuide – Bean Information window, do the following:
   a. If you want a name other than the actual feature name to be displayed for the property in the Visual Composition Editor, specify the name in the **Display name** field. This name appears when the property is listed in connection menus, the bean property sheet, and other windows.
   b. If you want a description other than the feature name to be displayed for the property in the Visual Composition Editor, specify the description in the **Short description** field. This description appears in certain windows, such as connection windows and the Promote Features window, when the property is selected.
   c. If you do not want the property to appear in development windows unless the user chooses to display expert features, select the **expert** check box.
   d. If you do not want the property to be exposed to the bean consumer, select the **hidden** check box.
   e. If you want to provide customized initialization of the property, specify a property editor class.
   f. Select **Finish** to add the property. VisualAge generates the following:
      - Public methods for the feature in the bean class
      - Bean information code that describes the feature in the BeanInfo class for the bean

You can modify BeanInfo for the property in the **Information** pane of the BeanInfo page. If you want the property to appear as a preferred feature in the connection menu of the bean, set the feature's *Preferred* property to *true*. If you do not want the property to appear in the property sheet of the bean, set the feature's *Design time property* property to *false*.

RELATED CONCEPTS

"Feature Naming Guidelines" on page 24

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

"Creating and Modifying a BeanInfo Class" on page 132

"Specifying Expert Features" on page 138

"Specifying Hidden Features" on page 138

RELATED REFERENCES

"New Property Feature SmartGuide" on page 250

"Bean Information SmartGuide" on page 249

"Features Pane—BeanInfo Page" on page 238

Adding Method Features

Define method features to represent bean behaviors or functions that you want other beans to have access to.

Add a new method feature in the BeanInfo page as follows:

1. From the tool bar, select **New Method Feature**. If you prefer, you can select **New Method Feature** from the **Features** menu. Either selection opens the SmartGuide – New Method Feature window.
2. In the SmartGuide – New Method Feature window, do the following:
   a. Specify the method name in the **Method name** field.
   b. Specify the method return type in the **Return type** field.
   c. If your method feature requires parameter input, specify the number of parameters in the **Parameter count** field.
   d. Select **Next** to open the either the SmartGuide – Parameter window or the SmartGuide – Bean Information window.
3. In the SmartGuide – Parameter window for each parameter, do the following:
   a. Specify the parameter name in the **Parameter name** field.
   b. Specify the parameter type in the **Parameter type** field.
   c. If you want a name other than the actual parameter name to be displayed for the parameter in the Visual Composition Editor, specify the name in the **Display name** field. This name appears when the parameter is listed in visual composition windows.
   d. If you want a description other than the feature name to be displayed for the parameter in the Visual Composition Editor, specify the description in the **Short description** field. This description appears when the parameter is selected in visual composition windows.

e. Select **Next** to open the SmartGuide – Bean Information window.

4. In the SmartGuide – Bean Information window, do the following:
   a. If you want a name other than the actual feature name to be displayed for the method in the Visual Composition Editor, specify the name in the **Display name** field. This name appears when the method is listed in connection menus, the Promote Features window, and other windows.
   b. If you want a description other than the feature name to be displayed for the method in the Visual Composition Editor, specify the description in the **Short description** field. This description appears in certain windows, such as connection windows and the Promote Features window, when the method is selected.
   c. If you do not want the method to appear in development windows unless the user chooses to display expert features, select the **expert** check box.
   d. If you do not want the method to be exposed to the bean consumer, select the **hidden** check box.
   e. Select **Finish** to add the method. VisualAge generates the following:
      - A public method for the feature in the bean class
      - Bean information code that describes the feature in the BeanInfo class for the bean

You can modify BeanInfo for the method in the **Information** pane of the BeanInfo page. If you want the method to appear as a preferred feature in the connection menu of the bean, set the feature’s *Preferred* property to *true*.

RELATED CONCEPTS

“Feature Naming Guidelines” on page 24

“Chapter 9. Generated Code” on page 33

RELATED TASKS

“Chapter 27. Defining Bean Interfaces for Visual Composition” on page 131

“Creating and Modifying a BeanInfo Class” on page 132

“Specifying Expert Features” on page 138

“Specifying Hidden Features” on page 138

RELATED REFERENCES

“New Method Feature SmartGuide” on page 253

“Parameter SmartGuide” on page 254

“Bean Information SmartGuide” on page 249

“Features Pane—BeanInfo Page” on page 238

Adding Event Features

Define event features to represent the occurrence of any events in your bean that you want other beans to be aware of.

You can add an event feature in the BeanInfo page based on either an existing event set or a new event set that you define. An *event set* consists of an event listener interface with associated event object and multicaster classes. The multicaster enables multiple listeners for an event.

Add an event feature based on an existing event set in the BeanInfo page as follows:

1. From the tool bar, select **New Event Set Feature**. If you prefer, you can select **New Event Set Feature** from the **Features** menu. Either selection opens the SmartGuide – New Event Set Feature window.
2. In the SmartGuide – New Event Set Feature window, do the following:
   a. Specify the event name in the **Event name** field.
   b. Select an event listener in the **Event listener** list.
   c. Select **Next** to open the SmartGuide – Bean Information window.
3. In the SmartGuide – Bean Information window, do the following:
   a. If you want a name other than the actual feature name to be displayed for the event in the Visual Composition Editor, specify the name in the **Display name** field. This name appears when the event is listed in connection menus, the Promote Features window, and other windows.
   b. If you want a description other than the feature name to be displayed for the event in the Visual Composition Editor, specify the description in the **Short description** field. This description appears in certain windows, such as connection windows and the Promote Features window, when the event is selected.
   c. If you do not want the event to appear in development windows unless the user chooses to display expert features, select the **expert** check box.
   d. If you do not want the event to be exposed to the bean consumer, select the **hidden** check box.
   e. Select **Finish** to add the method. VisualAge generates the following:
      - A public method for the feature in the bean class
      - Bean information code that describes the feature in the BeanInfo class for the bean

Add an event feature based on a new event set in the BeanInfo page as follows:

1. Select **New Listener Interface** from the **Features** menu to open the SmartGuide – New Event Listener window.
2. In the SmartGuide – New Event Listener window, do the following:
   a. Specify the event name in the **Event name** field.
   b. Specify the event listener name in the **Event listener** field. A default name is produced based on the name you specify in the **Event name** field.
   c. Specify the event object name in the **Event object** field. A default name is produced based on the name you specify in the **Event name** field.
   d. Specify the event multicaster name in the **Event Multicaster** field. A default name is produced based on the name you specify in the **Event name** field.
   e. Select **Next** to open the SmartGuide – Event Listener Methods window.
3. In the SmartGuide – Event Listener Methods window, do the following:
   a. For each method that you want to add to the listener, specify the method in the **Method name** field. Then, select the **Add** button. These listener methods respond to the event. You must add code that responds to the event in each method.
   b. Select **Next** to open the SmartGuide – Bean Information window.
4. In the SmartGuide – Bean Information window, do the following:
   a. If you want a name other than the actual feature name to be displayed for the event in the Visual Composition Editor, specify the name in the **Display**

name field. This name appears when the event is listed in connection menus, the Promote Features window, and other windows.

b. If you want a description other than the feature name to be displayed for the event in the Visual Composition Editor, specify the description in the **Short description** field. This description appears in certain windows, such as connection windows and the Promote Features window, when the event is selected.

c. If you do not want the event to appear in development windows unless the user chooses to display expert features, select the **expert** check box.

d. If you do not want the event to be exposed to the bean consumer, select the **hidden** check box.

e. Select **Finish** to add the method. VisualAge generates the following:

   - A public method for the feature in the bean class
   - Bean information code that describes the feature in the BeanInfo class for the bean

You can modify BeanInfo for the event in the **Information** pane of the BeanInfo page. If you want the event to appear as a preferred feature in the connection menu of the bean, set the feature's *Preferred* property to *true*.

RELATED CONCEPTS

"Feature Naming Guidelines" on page 24

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

"Creating and Modifying a BeanInfo Class" on page 132

"Specifying Expert Features" on page 138

"Specifying Hidden Features" on page 138

RELATED REFERENCES

"New Event Set Feature SmartGuide" on page 252

"New Event Listener SmartGuide" on page 251

"Event Listener Methods SmartGuide" on page 252

"Bean Information SmartGuide" on page 249

"Features Pane—BeanInfo Page" on page 238

Promoting Features of Embedded Beans

Promote features in the Visual Composition Editor as follows:

1. From the pop-up menu of the embedded bean, select **Promote bean feature** to open the Promote Features window.
2. For each feature that you want to promote, do the following:

   a. Select **Method**, **Property**, or **Event** to filter promotable features in the features list box.

   b. In the features list box, select the feature you are promoting.

   c. Select the *>>* button. The feature is moved to the **Promoted features** list.

d. If you do not want to use the default name, double-click the feature name in the **Promote feature name** field. Then, edit the name to change it.

3. Select **OK** to close the Promote Features window.
4. Save the composite bean to incorporate the features you just promoted. If you run the test tool, the bean is automatically saved.

RELATED CONCEPTS

"Default Promoted Feature Names" on page 24

"Feature Naming Guidelines" on page 24

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

"Creating and Modifying a BeanInfo Class" on page 132

RELATED REFERENCES

"Chapter 44. Promote Features Window" on page 213

Specifying Expert Features

You can designate *expert* features that you do not normally want listed in development windows. These are features that are complex or easily misused.

To designate a feature as expert, do either of the following:

- When adding the feature, select the **expert** check box in the SmartGuide – Bean Information window.
- Edit the bean information in the **Information** pane of the BeanInfo page. Select the *Expert* property, then select *true* in the value column.

RELATED CONCEPTS

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

"Creating and Modifying a BeanInfo Class" on page 132

RELATED REFERENCES

"Bean Information SmartGuide" on page 249

"Information Pane—BeanInfo Page" on page 240

Specifying Hidden Features

You can designate *hidden* features that you do not want to be available for connections and property settings. These are features that you use within the bean for implementation that you do not want exposed.

To designate a feature as hidden, do either of the following:

- When adding the feature, select the **hidden** check box in the SmartGuide – Bean Information window.

- Edit the bean information in the **Information** pane of the BeanInfo page. Select the *Hidden* property, then select *true* in the value column.

RELATED CONCEPTS

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

"Creating and Modifying a BeanInfo Class" on page 132

RELATED REFERENCES

"Bean Information SmartGuide" on page 249

"Information Pane—BeanInfo Page" on page 240

Chapter 28. Beans for Visual Composition

VisualAge provides a wide range of beans that you can use to visually compose your own program elements. These include the following:

- Basic user interface beans from the Abstract Windowing Toolkit (AWT)
- Enhanced user interface beans from the Java Foundation Classes (JFC) library
- Factory and variable beans for dynamically creating and referencing bean instances

The following topics describe beans provided by IBM. When you create your own beans, you can add them to the palette. See the related task topic on modifying the palette.

User Interface Beans

VisualAge provides a set of user interface beans that you can use to compose an applet or application. Basic user interface beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project. AWT beans are in the *java.awt* and *java.applet* packages. Enhanced user interface beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project. Swing beans are in *com.sun.java.swing* and related packages.

Although Swing and AWT components can be mixed, it is inadvisable. For this reason, VisualAge does not allow you to drop AWT beans on Swing beans. Because you might want to add Swing beans to AWT beans that you created before Swing was available, VisualAge *does* allow you to drop Swing beans on AWT beans. You can morph the AWT beans to Swing beans when you are ready to convert completely to Swing.

The problem with mixing AWT and Swing beans arises from the fact that all AWT components have peer classes that are specific to the operating system, while most Swing components do not. Components with system peers are known as *heavyweight* components. Components without system peers are known as *lightweight* components. The only Swing heavyweight components are JApplet, JDialog, JFrame, and JWindow. Painting problems occur if heavyweight components are children of lightweight parents, because the heavyweight components always paint over lightweight components.

VisualAge provides its own BeanInfo classes for Swing and AWT beans. These BeanInfo classes are tailored for visual composition.

Swing beans have a LookAndFeel (L&F) architecture that specifies how Swing components appear and behave. On the Visual Composition Editor, Swing beans appear in the default, cross-platform Metal L&F implementation. To change the runtime appearance to the System L&F of the current platform, execute the following code in your *main* method before you construct your components:

```
String myLookAndFeel = com.sun.java.swing.UIManager.getSystemLookAndFeelClassName();
com.sun.java.swing.UIManager.setLookAndFeel(myLookAndFeel);
```

The *myLookAndFeel* can be the name of any class that implements *com.sun.java.swing.LookAndFeel* and is available on the current platform. This code changes only the runtime L&F implementation and not the images on the Visual Composition Editor.

VisualAge information about Swing and AWT beans supplements class information from JavaSoft. VisualAge reference topics for these beans provide links to JavaSoft class information. The following topics describe these beans:

- “Chapter 29. Applet Beans” on page 143
- “Chapter 30. Window Beans” on page 145
- “Chapter 31. Pane and Panel Beans” on page 151
- “Chapter 32. Table and Tree Beans” on page 159
- “Chapter 33. Text Beans” on page 161
- “Chapter 34. List and Slider Beans” on page 167
- “Chapter 35. Button Beans” on page 173
- “Chapter 36. Menu and Tool Bar Beans” on page 179

Factory and Variable Beans

VisualAge provides Factory and Variable beans that you can use to dynamically create and reference bean instances. The following topic describes these beans:

- “Chapter 37. Factory and Variable Beans” on page 189

RELATED CONCEPTS

“Chapter 2. How Classes and Beans Are Related” on page 3

RELATED TASKS

“Chapter 22. Using VisualAge Beans in Visual Composition” on page 89

“Chapter 21. Managing the Beans Palette” on page 85

Chapter 29. Applet Beans

Applets are programs that can be downloaded and run by a Java-enabled web browser. These programs are generally small and specialized. An applet runs in a web page on a client system, within bounds specified by the page markup. A Java applet operates within constraints that provide security from remote system intrusion.

VisualAge provides applet beans from Swing and AWT packages. A basic applet bean from the Abstract Windowing Toolkit (AWT) is provided in the *Java class libraries* project, in the *java.applet* package. An enhanced applet bean from the Java Foundation Classes (JFC) library is provided in the *JFC class libraries* project, in the *com.sun.java.swing* package. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide applets:

Bean	Description
“JApplet” on page 144 or “Applet”	A program that can run in a web browser

RELATED TASKS

“Composing an Applet” on page 89

RELATED REFERENCES

“Chapter 28. Beans for Visual Composition” on page 141

Applet

Palette category
None

Palette bean
None

Project
Java class libraries

Package
java.applet

Type Applet

An Applet-based bean provides the foundation for a program that can be downloaded and run in a Java-enabled web browser. To use this bean, specify Applet as the superclass for a new applet bean.

Use a JApplet bean, rather than an Applet bean, if you want to use Swing components in the applet. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Composing an Applet” on page 89

RELATED REFERENCES

“Chapter 29. Applet Beans”

“Chapter 28. Beans for Visual Composition” on page 141

JApplet

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JApplet

A JApplet-based bean provides the foundation for a program that can be downloaded and run in a Java-enabled web browser. To use this bean, specify *com.sun.java.swing.JApplet* as the superclass for a new applet bean.

The JApplet bean provides a content pane in which to place other components. The content pane provides logical separation of the applet from its child components. With the exception of a JMenuBar, user interface components are added to the content pane, which completely covers the JApplet bean in the Visual Composition Editor. The JApplet bean can be accessed from the Beans List. The default content pane, a JAppletContentPane, is represented in the Beans List as the child of the JApplet bean. You can delete the default content pane and replace it with another container component.

Use an Applet bean, rather than a JApplet bean, if you want to use AWT components in the applet. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Composing an Applet” on page 89

RELATED REFERENCES

“Chapter 29. Applet Beans” on page 143

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 30. Window Beans

Windows beans are the primary visual context for other user interface components. VisualAge provides window beans from Swing and AWT packages. Basic window beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project, in the *java.awt* package. Enhanced window beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project, in the *com.sun.java.swing* package. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide windows:

Bean	Description
“JDialog” on page 147 or “Dialog”	A custom dialog, typically a secondary window
“FileDialog” on page 146	A dialog for accessing the file system
“JFrame” on page 148 or “Frame” on page 146	A desktop window with a title bar, sizing borders, and sizing buttons
“JInternalFrame” on page 148	A frame that is a child of another Swing component
“JWindow” on page 149 or “Window” on page 150	A window without a title bar, sizing borders, and sizing buttons

RELATED TASKS

“Composing a Window” on page 92

RELATED REFERENCES

“Chapter 28. Beans for Visual Composition” on page 141

Dialog

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Dialog

Use a Dialog bean to provide a custom dialog. A dialog is typically used to display or gather information for a single purpose.

The Dialog bean supports a client component in which to place other components. The default client component is a Panel bean named ContentsPane. You can delete the default client and replace it with another container component.

Use a JDialog bean, rather than a Dialog bean, if you want to use Swing components in the dialog. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Composing a Window" on page 92

RELATED REFERENCES

"Chapter 30. Window Beans" on page 145

"Chapter 28. Beans for Visual Composition" on page 141

FileDialog

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type FileDialog

Use a FileDialog bean to provide a dialog for accessing the file system. The user can open or save files using this dialog. File dialogs are useful in stand-alone applications. File dialogs are typically not used in applets due to security constraints.

The FileDialog bean represents a system file dialog. The bean appears on the free-form surface as an icon because it cannot be composed.

RELATED TASKS

"Composing a Window" on page 92

RELATED REFERENCES

"Chapter 30. Window Beans" on page 145

"Chapter 28. Beans for Visual Composition" on page 141

Frame

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Frame

Use a Frame bean to provide a desktop window with a title bar, sizing borders, and sizing buttons. You can add beans to the frame to define menus and other user interface components.

The Frame bean supports a client component in which to place other components. The default client component is a Panel bean named ContentsPane. You can delete the default client and replace it with another container component.

Use a JFrame bean, rather than a Frame bean, if you want to use Swing components in the frame. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Composing a Window” on page 92

RELATED REFERENCES

“Chapter 30. Window Beans” on page 145

“Chapter 28. Beans for Visual Composition” on page 141

JDialog

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JDialog

Use a JDialog bean to provide a custom dialog. A dialog is typically used to display or gather information for a single purpose.

The JDialog bean provides a content pane in which to place other components. The content pane provides logical separation of the dialog from its child components. With the exception of a JMenuBar, user interface components are added to the content pane. The default content pane, a JDialogContentPane, is represented in the Beans List as the child of the JDialog bean. You can delete the default content pane and replace it with another container component.

Use a Dialog bean, rather than a JDialog bean, if you want to use AWT components in the dialog. Although Swing and AWT beans can be mixed, it is inadvisable. Alternatively, use a JOptionPane bean to create any of a variety of standard dialogs.

RELATED TASKS

“Composing a Window” on page 92

RELATED REFERENCES

“Chapter 30. Window Beans” on page 145

“Chapter 28. Beans for Visual Composition” on page 141

JFrame

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JFrame

Use a JFrame bean to provide a desktop window with a title bar, sizing borders, and sizing buttons. You can add beans to the frame and its content pane to define menus and other user interface components.

The JFrame bean provides a content pane in which to place other components. The content pane provides logical separation of the frame from its child components. With the exception of a JMenuBar, user interface components are added to the content pane. The default content pane, a JFrameContentPane, is represented in the Beans List as the child of the JFrame bean. You can delete the default content pane and replace it with another container component.

Use a Frame bean, rather than a JFrame bean, if you want to use AWT components in the frame. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Composing a Window” on page 92

RELATED REFERENCES

“JInternalFrame”

“Chapter 30. Window Beans” on page 145

“Chapter 28. Beans for Visual Composition” on page 141

JInternalFrame

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JInternalFrame

Use a JInternalFrame bean to provide a frame that is a child of a JDesktopPane bean. By contrast, a JFrame bean is a child of the desktop. The user can manipulate an internal frame within a desktop pane. For example, the user can maximize, minimize, resize, move, or close the internal frame.

The JInternalFrame bean provides a content pane in which to place other components. The content pane provides logical separation of the frame from its child components. With the exception of a JMenuBar, user interface components are added to the content pane. The default content pane, a JInternalFrameContentPane, is represented in the Beans List as the child of the JInternalFrame bean. You can delete the default content pane and replace it with another container component.

RELATED TASKS

"Composing a Window" on page 92

RELATED REFERENCES

"JFrame" on page 148

"Chapter 30. Window Beans" on page 145

"Chapter 28. Beans for Visual Composition" on page 141

JWindow

Palette category
Swing

Palette bean
⧉

Project
JFC class libraries

Package
com.sun.java.swing

Type JWindow

Use a JWindow bean to add a window without a title bar, sizing borders, and sizing buttons. This bean is suitable for a splash window that your application displays briefly at startup.

The JWindow bean provides a content pane in which to place other components. The content pane provides logical separation of the window from its child components. With the exception of a JMenuBar, user interface components are added to the content pane, which completely covers the JWindow bean in the Visual Composition Editor. The JWindow bean can be accessed from the Beans List. The default content pane, a JWindowContentPane, is represented in the Beans List as the child of the JWindow bean. You can delete the default content pane and replace it with another container component.

Use a Window bean, rather than a JWindow bean, if you want to use AWT components in the window. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Composing a Window" on page 92

RELATED REFERENCES

“Chapter 30. Window Beans” on page 145

“Chapter 28. Beans for Visual Composition” on page 141

Window

Palette category
None

Palette bean
None

Project
Java class libraries

Package
java.awt

Type Window

Use a Window bean to add a window without a title bar, sizing borders, and sizing buttons. This bean is suitable for a splash window that your application displays briefly at startup.

Use a JWindow bean, rather than a Window bean, if you want to use Swing components in the window. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Composing a Window” on page 92

RELATED REFERENCES

“Chapter 30. Window Beans” on page 145

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 31. Pane and Panel Beans

A pane or panel is a container for other components. It is used within another pane or panel, within a window, or within an applet. VisualAge provides pane and panel beans from Swing and AWT packages. Basic pane and panel beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project, in the *java.awt* package. Enhanced pane and panel beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project, in the *com.sun.java.swing* package. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide panes and panels:

Bean	Description
“JDesktopPane”	A pane for a desktop within another Swing container
“JEditorPane” on page 152	A pane for editing defined text types, such as HTML
“JOptionPane” on page 152	A simple dialog pane
“JPanel” on page 153 or “Panel” on page 156	A composition surface for user interface components
“JScrollPane” on page 153 or “ScrollPane” on page 156	A scrollable view for another component
“JSplitPane” on page 154	A split view for other components
“JTabbedPane” on page 155	A tabbed view for other components
“JTextPane” on page 155	A pane for editing text with visible styles and embedded objects

RELATED TASKS

“Adding a Pane or Panel” on page 95

RELATED REFERENCES

“Chapter 28. Beans for Visual Composition” on page 141

“Chapter 29. Applet Beans” on page 143

“Chapter 30. Window Beans” on page 145

JDesktopPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JDesktopPane

Use a JDesktopPane bean to provide a desktop within another Swing container. Add one or more JInternalFrame beans to the desktop pane.

You should not use a JDesktopPane bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Pane or Panel” on page 95

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

JEditorPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JEditorPane

Use a JEditorPane bean for editing defined text types, such as HTML.

You should not use a JEditorPane bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Pane or Panel” on page 95

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

JOptionPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JOptionPane

Use a JOptionPane bean to provide a simple dialog for an input prompt, a message, or user confirmation. The dialog is modal, so the thread is held until the user dismisses the dialog.

You should not use a JOptionPane bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Pane or Panel" on page 95

RELATED REFERENCES

"Chapter 31. Pane and Panel Beans" on page 151

"Chapter 28. Beans for Visual Composition" on page 141

JPanel

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JPanel

Use a JPanel bean as a composition surface for other user interface components, such as buttons, lists, and text. You can add a panel to a window, an applet, or another panel. In a Swing window or JApplet, a JPanel can either serve as the contentPane, or be added to the contentPane.

Use a Panel bean, rather than a JPanel bean, if you want to use AWT components in the panel. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Pane or Panel" on page 95

RELATED REFERENCES

"Chapter 31. Pane and Panel Beans" on page 151

"Chapter 28. Beans for Visual Composition" on page 141

JScrollPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JScrollPane

Use a JScrollPane bean to provide a pane with scroll bars. This enables you to define a pane that is not always completely within view. You can place one component in the scroll pane. If you want multiple components within the scroll pane, add a JPanel bean to the scroll pane and place the components on the panel.

Use a ScrollPane bean, rather than a JScrollPane bean, if you want to use AWT components in the pane. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Pane or Panel” on page 95

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

JSplitPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JSplitPane

Use a JSplitPane bean to provide a split view for other components. You can place one component on each pane. If you want multiple components within a pane, add a JPanel bean to the pane and place the components on the panel.

You should not use a JSplitPane bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Pane or Panel” on page 95

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

JTabbedPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JTabbedPane

Use a JTabbedPane bean to provide a tabbed view for other components. Each component you drop on a JTabbedPane becomes a new page with a separate tab. If you want multiple components within a page, add a JPanel bean to the scroll pane and place the components on the panel. When you add a JTabbedPane bean, a JPanel bean is automatically added as the first page.

You should not use a JTabbedPane bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Pane or Panel" on page 95

RELATED REFERENCES

"Chapter 31. Pane and Panel Beans" on page 151

"Chapter 28. Beans for Visual Composition" on page 141

JTextPane

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JTextPane

Use a JTextPane bean to provide a pane for editing text with visible styles and embedded objects.

You should not use a JTextPane bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Pane or Panel" on page 95

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

Panel

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Panel

Use a Panel bean as a composition surface for other user interface components, such as buttons, lists, and text. You can add a panel to a window, an applet, or another panel. In an AWT window, a Panel can either serve as the client component, or be added to the client component.

Use a JPanel bean, rather than a Panel bean, if you want to use Swing components in the panel. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Pane or Panel” on page 95

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

ScrollPane

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type ScrollPane

Use a ScrollPane bean to provide a pane with scroll bars. This enables you to define a pane that is not always completely within view. You can place one component in the scroll pane. If you want multiple components within the scroll pane, add a Panel bean to the scroll pane and place the components on the panel.

Use a JScrollPane bean, rather than a ScrollPane bean, if you want to use Swing components in the pane. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Pane or Panel" on page 95

RELATED REFERENCES

"Chapter 31. Pane and Panel Beans" on page 151

"Chapter 28. Beans for Visual Composition" on page 141

Chapter 32. Table and Tree Beans

A table or tree provides a view of objects from a data model that organizes objects in a tabular or expandable tree format. VisualAge provides table and tree beans from Swing packages in the *JFC class libraries* project, in the *com.sun.java.swing* and *com.sun.java.swing.table* packages. You should not use these beans with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide tables and trees:

Bean	Description
"JTable"	A table view of objects from a table data model
"TableColumn" on page 160	A view of objects from a table data model column
"JTree" on page 160	A tree view of objects from a tree data model

RELATED TASKS

"Adding a Table or Tree View" on page 99

RELATED REFERENCES

"Chapter 28. Beans for Visual Composition" on page 141

JTable

Palette category
Swing

Palette bean
◫

Project
JFC class libraries

Package
com.sun.java.swing

Type JTable

Use a JTable bean to provide a view of objects from a table data model. Use the table model to define or derive the objects for the table. You can do this either by coding the model or by using a tool such as the database builder to create the data model.

The user can select objects from the table, manipulate columns, and directly edit table cells.

You should not use a JTable bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Table or Tree View" on page 99

RELATED REFERENCES

"Chapter 32. Table and Tree Beans"

"Chapter 28. Beans for Visual Composition" on page 141

TableColumn

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing.table

Type TableColumn

Use a TableColumn bean to provide a view of objects from a table data model column. This bean enables you to map a column in a JTable bean to a column in the table data model. You can also use it to define visual properties of a table column. If you want a default table view of each column in the data model, do not add any TableColumn beans to a JTable bean.

RELATED TASKS

“Adding a Table or Tree View” on page 99

RELATED REFERENCES

“Chapter 32. Table and Tree Beans” on page 159

“Chapter 28. Beans for Visual Composition” on page 141

JTree

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JTree

Use a JTree bean to provide a view of objects from a tree data model. Use the tree model to define or derive the objects for the tree.

You should not use a JTree bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Table or Tree View” on page 99

RELATED REFERENCES

“Chapter 32. Table and Tree Beans” on page 159

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 33. Text Beans

Text components are available for simple text and for enhanced text and editing panes. VisualAge provides text beans from Swing and AWT packages. Basic text beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project, in the *java.awt* package. Enhanced text beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project, in the *com.sun.java.swing* and *com.sun.java.swing.text* packages. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide simple text components:

Bean	Description
“JLabel” or “Label” on page 163	A label, usually to identify another component
“JPasswordField” on page 162	A text field for sensitive data
“JTextArea” on page 162 or “TextArea” on page 164	A multiline text area
“JTextField” on page 163 or “TextField” on page 164	A single-line text field

RELATED TASKS

“Adding a Text Component” on page 101

RELATED REFERENCES

“Chapter 31. Pane and Panel Beans” on page 151

“Chapter 28. Beans for Visual Composition” on page 141

JLabel

Palette category
Swing

Palette bean
Ab

Project
JFC class libraries

Package
com.sun.java.swing

Type JLabel

Use a JLabel bean to provide a text label for your user interface. You can add a graphic to the label as well as text. You can also define a mnemonic in the label as a means of quick access to a text input field.

Use a Label bean, rather than a JLabel bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Text Component” on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

JPasswordField

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JPasswordField

Use a JPasswordField bean to provide a text field for sensitive data. A JPasswordField is a text field that always uses an echo character to mask characters that are entered. The echo character can be specified in the property sheet for the JPasswordField. You can use a focus accelerator to provide quick accessability to the password field.

You should not use a JPasswordField bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Text Component" on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

JTextArea

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JTextArea

Use a JTextArea bean to provide a large area for text entry or presentation. A text area can contain multiple lines of text. To make a JTextArea bean scrollable, drop it in a JScrollPane. You can use a focus accelerator to provide quick accessability to the text area.

Use a TextArea bean, rather than a JTextArea bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Text Component" on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

JTextField

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JTextField

Use a JTextField bean to provide a field for text entry or presentation. A text field consists of a single line. You can use a focus accelerator to provide quick accessability to the text field.

Use a TextField bean, rather than a JTextField bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Text Component" on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

Label

Palette category
AWT

Palette bean

Ab

Project
Java class libraries

Package
java.awt

Type Label

Use a Label bean to provide a text label for your user interface.

Use a JLabel bean, rather than a Label bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Text Component" on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

TextArea

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type TextArea

Use a TextArea bean to provide a large area for text entry or presentation. A text area can contain multiple lines, and is vertically and horizontally scrollable.

Use a JTextArea bean, rather than a TextArea bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Text Component" on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

TextField

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type TextField

Use a TextField bean to provide a field for text entry or presentation. A text field consists of a single line.

Use a JTextField bean, rather than a TextField bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Text Component" on page 101

RELATED REFERENCES

"Chapter 33. Text Beans" on page 161

"Chapter 28. Beans for Visual Composition" on page 141

Chapter 34. List and Slider Beans

List components provide a list of items for the user to select. Slider components show a range of selection values or show progress for the duration of an operation. VisualAge provides list and slider beans from Swing and AWT packages. Basic list and slider beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project, in the *java.awt* package. Enhanced list and slider beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project, in the *com.sun.java.swing* package. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide visual list and slider components:

Bean	Description
"JComboBox" on page 168 or "Choice"	A selectable list with an entry field
"JList" on page 168 or "List" on page 170	A selectable list of choices
"JProgressBar" on page 169	A progress indicator
"JScrollBar" on page 169 or "Scrollbar" on page 171	A scrolling component
"JSlider" on page 170	A selection component for a range of values

RELATED TASKS

"Adding a List or Slider Component" on page 104

RELATED REFERENCES

"Chapter 28. Beans for Visual Composition" on page 141

Choice

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Choice

Use a Choice bean to provide a drop-down list that lets the user select a single choice. The current choice is always displayed. The user can open the list by selecting the drop-down button.

Use a JComboBox bean, rather than a Choice bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a List or Slider Component" on page 104

RELATED REFERENCES

"Chapter 34. List and Slider Beans" on page 167

"Chapter 28. Beans for Visual Composition" on page 141

JComboBox

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JComboBox

Use a JComboBox bean to provide a selectable drop-down list. You can choose to allow direct text entry as well as selection from the list.

Use a Choice bean, rather than a JComboBox bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a List or Slider Component" on page 104

RELATED REFERENCES

"Chapter 34. List and Slider Beans" on page 167

"Chapter 28. Beans for Visual Composition" on page 141

JList

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JList

Use a JList bean to provide a list of choices from which the user can make one or more selections. Your program can add or remove choices from the list. If you want the list to be scrollable, you should place it in a JScrollPane.

By default, the user can select one choice from the list. When the user selects a choice, any previously selected choice is no longer selected. You can change the behavior of the list to allow multiple selection.

Use a List bean, rather than a JList bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a List or Slider Component” on page 104

RELATED REFERENCES

“Chapter 34. List and Slider Beans” on page 167

“Chapter 28. Beans for Visual Composition” on page 141

JProgressBar

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JProgressBar

Use a JProgressBar bean to provide a progress indicator for an operation.

You should not use a JProgressBar bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a List or Slider Component” on page 104

RELATED REFERENCES

“Chapter 34. List and Slider Beans” on page 167

“Chapter 28. Beans for Visual Composition” on page 141

JScrollBar

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JScrollBar

Use a JScrollBar bean to provide a scrolling component. Generally, a JScrollPane is a suitable alternative for a scrollable view with scroll bars.

Use a Scrollbar bean, rather than a JScrollBar bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a List or Slider Component" on page 104

RELATED REFERENCES

"Chapter 34. List and Slider Beans" on page 167

"Chapter 28. Beans for Visual Composition" on page 141

JSlider

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JSlider

Use a JSlider bean to provide a selection component for a range of values.

You should not use a JSlider bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a List or Slider Component" on page 104

RELATED REFERENCES

"Chapter 34. List and Slider Beans" on page 167

"Chapter 28. Beans for Visual Composition" on page 141

List

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type List

Use a List bean to provide a list of choices from which the user can make one or more selections. Your program can add or remove choices from the list.

By default, the user can select one choice from the list. When the user selects a choice, any previously selected choice is no longer selected. You can change the behavior of the list to allow multiple selection.

Use a JList bean, rather than a List bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a List or Slider Component” on page 104

RELATED REFERENCES

“Chapter 34. List and Slider Beans” on page 167

“Chapter 28. Beans for Visual Composition” on page 141

Scrollbar

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Scrollbar

Use a Scrollbar bean to provide a slider that the user can manipulate to select a value from a range of values. A *scroll bar* consists of a scroll shaft that represents the range of values, a scroll box within the range, and scroll arrows at the ends of the range. A scroll bar can be either horizontal or vertical.

Use a Scrollbar bean, rather than a JScrollBar bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a List or Slider Component” on page 104

RELATED REFERENCES

“Chapter 34. List and Slider Beans” on page 167

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 35. Button Beans

VisualAge provides button beans from Swing and AWT packages. Basic button beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project, in the *java.awt* package. Enhanced button beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project, in the *com.sun.java.swing* package. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide button components:

Bean	Description
"JButton" on page 175 or "Button"	A push button, generally used to perform a function
"JCheckBox" on page 175 or "Checkbox" on page 174	A setting button that is checked when selected
"JRadioButton" on page 176 or "CheckboxGroup" on page 174	A radio button or group for mutually exclusive settings
"JToggleButton" on page 176	A two-state push button that appears to be pushed in when selected

RELATED TASKS

"Adding a Button Component" on page 107

RELATED REFERENCES

"Chapter 28. Beans for Visual Composition" on page 141

Button

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Button

Use a Button bean to provide a push button that the user can select to perform an action. For example, you can define an **OK** button to let the user save changes and close a window.

Use a JButton bean, rather than a Button bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Button Component" on page 107

RELATED REFERENCES

"Chapter 35. Button Beans"

“Chapter 28. Beans for Visual Composition” on page 141

Checkbox

Palette category
AWT

Palette bean
☑

Project
Java class libraries

Package
java.awt

Type Checkbox

Use a Checkbox bean to provide a settings choice that has two states, such as on and off. A mark in the check box indicates that the choice is selected.

Check boxes are independently selectable, unless they are defined in a group. Use check boxes in a group to provide a set of mutually exclusive choices that appear as radio buttons. To define a group, associate each Checkbox bean with a CheckboxGroup bean.

Use a JCheckBox bean, rather than a Checkbox bean, if you want to use it in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Button Component” on page 107

RELATED REFERENCES

“Chapter 35. Button Beans” on page 173

“Chapter 28. Beans for Visual Composition” on page 141

CheckboxGroup

Palette category
AWT

Palette bean
◉

Project
Java class libraries

Package
java.awt

Type CheckboxGroup

Use a CheckboxGroup bean to provide a set of mutually exclusive choices that appear as radio buttons. Each choice in the group is a Checkbox bean that you associate with the CheckboxGroup bean.

Use JRadioButton beans with a Swing ButtonGroup, rather than Checkbox beans with a CheckboxGroup, if you want to use them in a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Button Component" on page 107

RELATED REFERENCES

"Chapter 35. Button Beans" on page 173

"Chapter 28. Beans for Visual Composition" on page 141

JButton

Palette category
Swing

Palette bean
▭

Project
JFC class libraries

Package
com.sun.java.swing

Type JButton

Use a JButton bean to provide a push button that the user can select to perform an action. For example, you can define an **OK** button to let the user save changes and close a window.

Use a Button bean, rather than a JButton bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Button Component" on page 107

RELATED REFERENCES

"Chapter 35. Button Beans" on page 173

"Chapter 28. Beans for Visual Composition" on page 141

JCheckBox

Palette category
Swing

Palette bean
☑

Project
JFC class libraries

Package
com.sun.java.swing

Type JCheckBox

Use a JCheckBox bean to provide a settings choice that has two states, such as on and off. A mark in the check box indicates that the choice is selected. You can customize the images used for unselected and selected check boxes.

Use a Checkbox bean, rather than a JCheckBox bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Button Component" on page 107

RELATED REFERENCES

"Chapter 35. Button Beans" on page 173

"Chapter 28. Beans for Visual Composition" on page 141

JRadioButton

Palette category
Swing

Palette bean
⊙

Project
JFC class libraries

Package
com.sun.java.swing

Type JRadioButton

Use a JRadioButton bean to provide one of a group of mutually exclusive settings choices. A mark on the radio button indicates that the choice is selected.

Use a Checkbox bean with a CheckboxGroup bean, rather than a JRadioButton bean, if you want to use it in an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Button Component" on page 107

RELATED REFERENCES

"Chapter 35. Button Beans" on page 173

"Chapter 28. Beans for Visual Composition" on page 141

JToggleButton

Palette category
Swing

Palette bean
▣

Project
JFC class libraries

Package
com.sun.java.swing

Type JToggleButton

Use a JToggleButton bean to provide a two-state push button. The button appears to be pushed in when selected, and popped out when not selected. Use JToggleButton beans in a button group for a set of mutually exclusive functions. When the user selects an unselected button in the group, the previously selected button is popped out.

You should not use a JToggleButton bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Button Component” on page 107

RELATED REFERENCES

“Chapter 35. Button Beans” on page 173

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 36. Menu and Tool Bar Beans

VisualAge provides menu beans from Swing and AWT packages and tool bar beans from Swing and IBM packages. Basic menu beans from the Abstract Windowing Toolkit (AWT) are provided in the *Java class libraries* project, in the *java.awt* package. Enhanced menu and tool bar beans from the Java Foundation Classes (JFC) library are provided in the *JFC class libraries* project, in the *com.sun.java.swing* package. Visual composition implementation beans are provided in the *IBM Java Implementation* project, in the *com.ibm.uvm.abt.edit* package. Although Swing and AWT beans can be mixed, it is inadvisable.

The following beans provide menu and tool bar components:

Bean	Description
“JCheckBoxMenuItem” on page 180 or “CheckboxMenuItem”	A menu choice that toggles a setting on and off
“JMenu” on page 180 or “Menu” on page 185	A cascade menu for another menu
“JMenuBar” on page 181 or “MenuBar” on page 186	A menu bar for a window
“JMenuItem” on page 182 or “MenuItem” on page 187	A menu choice that calls a method
“JPopupMenu” on page 182 or “PopupMenu” on page 188	A pop-up menu for window components
“JRadioButtonMenuItem” on page 183	A menu choice that provides one of a set of mutually exclusive setting values
“JSeparator” on page 183 or “MenuSeparator” on page 187	A horizontal line that separates groups of related choices
“JToolBar” on page 184	A graphical set of tool choices
“JToolBarButton” on page 184	A button for a tool bar
“JToolBarSeparator” on page 185	A visual separator between components in a tool bar

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 28. Beans for Visual Composition” on page 141

CheckboxMenuItem

Palette category
AWT

Palette bean
☑

Project
Java class libraries

Package
java.awt

Type CheckboxMenuItem

Use a CheckboxMenuItem bean to provide a menu setting choice that the user can toggle on and off.

Use a JCheckBoxMenuItem bean, rather than a CheckboxMenuItem bean, if you want to use it in a Swing menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JCheckBoxMenuItem

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JCheckBoxMenuItem

Use a JCheckBoxMenuItem bean to provide a menu setting choice that the user can toggle on and off.

Use a CheckboxMenuItem bean, rather than a JCheckBoxMenuItem bean, if you want to use it in an AWT menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JMenu

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JMenu

Use a JMenu bean to provide a set of related choices for a window or component. You can add the menu either as a pull-down menu for a menu bar or as a cascade menu for another menu. You can also add subclasses of AbstractAction to the free-form surface and add them to the menu in the menu get method.

Use a Menu bean, rather than a JMenu bean, if you want to use it with an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

JMenuBar

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JMenuBar

Use a JMenuBar bean to provide a set of pull-down menus for a window. When you add a JMenuBar bean, one menu is automatically added to the menu bar. You can modify this menu and add other menus that you need.

Use a MenuBar bean, rather than a JMenuBar bean, if you want to use it with an AWT window or applet. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

JMenuItem

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JMenuItem

Use a JMenuItem bean to provide a functional choice for a menu.

Use a MenuItem bean, rather than a JMenuItem bean, if you want to use it in an AWT menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JPopupMenu

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JPopupMenu

Use a JPopupMenu bean to provide a pop-up list of choices for window components. The user can display the menu by clicking a pop-up mouse button on a component. You can also add subclasses of AbstractAction to the free-form surface and add them to the menu in the menu get method.

Use a PopupMenu bean, rather than a JPopupMenu bean, if you want to use it with an AWT container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JRadioButtonMenuItem

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JRadioButtonMenuItem

Use a JRadioButtonMenuItem bean to provide a menu setting choice for a group of mutually exclusive choices.

You should not use a JRadioButtonMenuItem bean in an AWT menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JSeparator

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JSeparator

Use a JSeparator bean to visually separate other components. It is commonly used to draw a horizontal line between groups of related choices in a menu, but can also be used to separate components on a panel.

Use a MenuSeparator bean, rather than a JSeparator bean, if you want to use it in an AWT menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JToolBar

Palette category
Swing

Palette bean

Project
JFC class libraries

Package
com.sun.java.swing

Type JToolBar

Use a JToolBar bean to provide a graphical set of tool choices. When you add a JToolBar bean, a JToolBarButton bean is automatically added as the first component. Add JToolBarButtons and other components to the tool bar. You can also add subclasses of AbstractAction to the free-form surface and add them to the tool bar in the tool bar get method.

You should not use a JToolBar bean with AWT components. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

JToolBarButton

Palette category
Swing

Palette bean

Project
IBM Java Implementation

Package
com.ibm.uvm.abt.edit

Type JToolBarButton

Use a JToolBarButton bean to provide a JButton bean for a tool bar. VisualAge customizes the JButton bean for a tool bar by presetting certain properties as follows:

Property	Setting
icon	question mark
text	null
margin	0,0,0,0
horizontalTextPosition	*CENTER*
verticalTextPosition	*BOTTOM*

The JToolBarButton bean is provided as a convenience. VisualAge generates code for a JButton bean.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

JToolBarSeparator

Palette category
Swing

Palette bean

Java class
com.ibm.uvm.abt.edit.JToolBarSeparator

Project
IBM Java Implementation

Package
com.ibm.uvm.abt.edit

Type JToolBarSeparator

Use a JToolBarSeparator bean to provide visual separation between other components on a tool bar. You can drop a JToolBarSeparator bean only on a JToolBar bean. The JToolBarSeparator bean is provided as a convenience. VisualAge generates code for the addSeparator() method of the JToolBar bean.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

Menu

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type Menu

Use a Menu bean to provide a set of related choices for a window or component. You can add the menu either as a pull-down menu for a menu bar or as a cascade menu for another menu.

Use a JMenu bean, rather than a Menu bean, if you want to use it with a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

MenuBar

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type MenuBar

Use a MenuBar bean to provide a set of pull-down menus for a window. When you add a MenuBar bean, one menu is automatically added to the menu bar. You can modify this menu and add other menus that you need.

Use a JMenuBar bean, rather than a MenuBar bean, if you want to use it with a Swing window or applet. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

MenuItem

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type MenuItem

Use a MenuItem bean to provide a functional choice for a menu.

Use a JMenuItem bean, rather than a MenuItem bean, if you want to use it in a Swing menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

MenuSeparator

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type MenuSeparator

Use a MenuSeparator bean to provide a horizontal line between groups of related menu choices.

Use a JSeparator bean, rather than a MenuSeparator bean, if you want to use it in a Swing menu. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

“Adding a Menu or Tool Bar” on page 109

RELATED REFERENCES

“Chapter 36. Menu and Tool Bar Beans” on page 179

“Chapter 28. Beans for Visual Composition” on page 141

PopupMenu

Palette category
AWT

Palette bean

Project
Java class libraries

Package
java.awt

Type PopupMenu

Use a PopupMenu bean to provide a pop-up list of choices for window components. The user can display the menu by clicking a pop-up mouse button on a component.

Use a JPopupMenu bean, rather than a PopupMenu bean, if you want to use it with a Swing container. Although Swing and AWT beans can be mixed, it is inadvisable.

RELATED TASKS

"Adding a Menu or Tool Bar" on page 109

RELATED REFERENCES

"Chapter 36. Menu and Tool Bar Beans" on page 179

"Chapter 28. Beans for Visual Composition" on page 141

Chapter 37. Factory and Variable Beans

VisualAge provides beans that enable you to dynamically create and reference bean instances visually. These beans are not Java classes. A Factory creates new instances of a bean type. A Variable references any instance of a bean type that you assign to it. With either a Factory or a Variable, you specify the bean type that it can create or reference.

The following beans provide support for bean instances:

Bean	Description
“Factory”	A bean that dynamically creates instances of Java beans
“Variable” on page 190	A bean that provides access to instances of Java beans

RELATED TASKS

“Dynamically Creating and Accessing a Bean Instance” on page 113

RELATED REFERENCES

“Chapter 28. Beans for Visual Composition” on page 141

Factory

Palette category
Other

Palette bean

Project
None

Package
None

Type None

A Factory bean dynamically creates instances of a Java bean that you specify. A Factory is not a Java class, and is not an instance of the bean it creates. The Factory creates a bean instance whenever an event occurs, based on a connection you make from the event.

Generally, when you add a bean to a composite bean, a fixed instance of that bean is created. When you add a Factory bean, however, a bean instance is not created. Instead, you can make a connection to dynamically create a bean instance of the type specified for the Factory. Connections to the Factory’s features then operate on the created bean instance. If you create another bean instance, connections to the Factory’s features operate on the newly created bean instance rather than on the previously created instance. The Factory bean serves as a bean instance generator.

RELATED TASKS

“Dynamically Creating and Accessing a Bean Instance” on page 113

RELATED REFERENCES

“Chapter 37. Factory and Variable Beans” on page 189

“Chapter 28. Beans for Visual Composition” on page 141

Variable

Palette category
Other

Palette bean
[ ]

Project
None

Package
None

Type None

Use a Variable bean to reference any instance of a particular class. A Variable is not a Java class, and is not an instance of the class it represents. Variable beans are commonly used to represent tear-off properties and objects from other composite beans.

Generally, when you add a bean to a composite bean, a single fixed instance of that bean is created. When you add a Variable bean, however, a bean instance is not created. Instead, you can make a connection to assign any bean instance of the type specified for the Variable. Connections to the Variable’s features then operate on the assigned bean instance. If you assign another bean instance to the Variable, connections to the Variable’s features operate on the newly assigned bean instance rather than on the previously assigned instance. The Variable bean serves as a reference for bean instances.

RELATED TASKS

“Dynamically Creating and Accessing a Bean Instance” on page 113

RELATED REFERENCES

“Chapter 37. Factory and Variable Beans” on page 189

“Chapter 28. Beans for Visual Composition” on page 141

Chapter 38. Visual Composition Editor

The Visual Composition Editor is a powerful composing tool that you can use to:

- Build the user interface for your program by dropping beans.
- Construct business logic by connecting the beans.
- Edit existing beans.

The Visual Composition Editor makes it easy to build applets, beans, and entire applications using the functions available on the menu bar, pop-up menus, tool bar, and the variety of reusable beans on the beans palette. A description of the functions on the tool bar or beans palette appears when the mouse pointer is positioned over the item.

Areas in this window

- “Free-Form Surface” on page 8
- “Beans Palette” on page 9
- “Chapter 39. The Menu Bar in Visual Composition” on page 195
- “The Tool Bar in Visual Composition”
- “Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201
- “Status Area—Visual Composition Editor”

RELATED CONCEPTS

“Adding Beans in the Visual Composition Editor” on page 10

RELATED TASKS

“Chapter 16. Working with Beans Visually” on page 53

“Chapter 19. Connecting Beans” on page 77

“Chapter 17. Composing Beans Visually” on page 67

“Saving a Bean” on page 69

RELATED REFERENCES

“Properties” on page 196

“Chapter 39. The Menu Bar in Visual Composition” on page 195

“Chapter 41. Pop-Up Menus for the Visual Composition Editor” on page 201

“Chapter 40. Keys” on page 199

“Chapter 56. BeanInfo Page” on page 237

Status Area—Visual Composition Editor

The status area displays information about the last operation or your current selection.

The Tool Bar in Visual Composition

Use the tools on the tool bar to help you build the user interface for your program.

You can also access some or all of these tools from the **Tools** pull-down menu in the Visual Composition Editor, and from pop-up menus.

The alignment tools are only available when using the null layout manager.

The anchor bean, indicated by solid selection handles, is the bean that serves as the alignment reference. When you want to align beans with one another, select the ones you want to move and select the anchor bean last. You can also change the anchor bean by holding the Ctrl key and double-clicking on the new anchor bean.

Tools in this area

“Run” on page 196

“Properties” on page 196

“Beans List” on page 196

“Show Connections” on page 196

“Hide Connections” on page 196

“Align Left” on page 197

“Align Center” on page 197

“Align Right” on page 197

“Align Top” on page 197

“Align Middle” on page 197

“Align Bottom” on page 197

“Distribute Horizontally” on page 197

“Distribute Vertically” on page 197

“Match Width” on page 197

“Match Height” on page 198

RELATED CONCEPTS

“Chapter 9. Generated Code” on page 33

“Property Sheets” on page 11

“Chapter 8. Connections” on page 27

RELATED TASKS

“Positioning Beans” on page 72

“Running and Testing Beans” on page 69

“Saving a Bean” on page 69

“Chapter 17. Composing Beans Visually” on page 67

“Moving Beans” on page 74

“Chapter 19. Connecting Beans” on page 77

“Working in the Beans List” on page 65

“Editing Bean Labels” on page 60

“Renaming Beans and Connections” on page 53

“Deleting Beans” on page 75

“Showing and Hiding Connections” on page 81

RELATED REFERENCES

“Layout” on page 204

“Connect” on page 203

Chapter 39. The Menu Bar in Visual Composition

The following menus provide options unique to the Visual Composition Editor:

- “Bean”
- “Tools”

Bean

The **Bean** menu provides options to perform the following tasks:

Save Bean	Saves the current bean and generates and compiles its code.
Re-generate Code	Generates and compiles the code for the current bean.
Run—In Applet Viewer	Saves the current bean, generates and compiles its code, and runs the bean in an applet viewer.
Run—Run main	Saves the current bean, generates and compiles its code, and runs the bean in a test frame.
Run—Set Class Path	Modifies the path for the current class. VisualAge adds the path for each dropped bean. However, if you are using other files, such as .gif files, be sure to add their location to the path.
Modify Palette	Manipulates categories and beans in the beans palette.
Fix Unresolved References	Changes the class of an unknown component to one that is loaded into your workspace. Before you select this, read “Chapter 53. Resolve Class References” on page 231 .
Construct Visuals from Source	Reverse-engineers a visual composite from Java source code. Before you select this, read “Chapter 6. Use of Visual Beans Created in Other Tools” on page 21.

Tools

Select **Tools** to help build and manipulate your user interface. The Visual Composition Editor also provides such tools as **Show connections** and **Hide connections** that lets you change how the free-form surface looks as you build your user interface.

You can access all or some of these tools from the tool bar, which is just below the menu bar in the Visual Composition Editor, and from pop-up menus.

The alignment tools are only available when using the null layout manager.

The anchor bean, indicated by solid selection handles, is the bean that serves as the alignment reference. When you want to align beans with one another, select the ones you want to move and select the anchor bean last. You can also change the anchor bean by holding the Ctrl key and double-clicking on the new anchor bean.

Run

Select **Run** to save the bean, generate code, compile the class, and run the compiled bean in an applet window.

Properties

Select **Properties** to display the property sheet for the bean you selected.

The property sheet contains editable values for the selected bean. You can open the property sheet for a selected bean either from the Visual Composition Editor or Beans List window.

If you selected multiple beans and then **Properties**, a property sheet appears and displays the common properties for the selected beans. When you change a property on the property sheet, the change affects all the selected beans.

Beans List

Select **Beans List** to display a list of the beans and connections used in your user interface. You can perform many of the same tasks within the beans list as you would in the Visual Composition Editor. This ability is particularly important when you are working with components that are covered by other components. For example, if your base bean is a panel using a border layout, another panel used as the center component expands to fill all empty space. Performing tasks on the covered border layout panel is difficult.

Show Connections

Select to display the connections you create between beans. This tool works on all connections only if nothing is selected. Otherwise, only the connections to and from the selected beans are affected.

The Visual Composition Editor displays all connections by default.

Hide Connections

Select **Hide connections** to conceal the connecting links between beans. If you wish to hide all connections, do not select any beans. Otherwise, select only the beans with connections you wish to hide.

If your program has numerous connections, you can reduce the visual clutter by hiding those not currently being modified.beans

Align Left

Select **Align left** to move the selected beans so that their left edges are aligned.

Align Center

Select **Align center** to move the selected beans so that their centers are aligned vertically.

Align Right

Select **Align right** to move the selected beans so that their right edges are aligned.

Align Top

Select **Align top** to move the selected beans so that their top edges are aligned.

Align Middle

Select **Align middle** to move the selected beans so that their centers are aligned horizontally.

Align Bottom

Select **Align bottom** to move the selected beans so that their bottom edges are aligned.

Distribute Horizontally

Select **Distribute horizontally** to move the selected beans so that they are spaced evenly between the left and right container borders.

Distribute Vertically

Select **Distribute vertically** to move the selected beans so that they are spaced evenly between the top and bottom of the container borders.

Match Width

Select

Match width to size the selected beans to the width of the anchor bean.

Match Height

Select **Match height** to size the selected beans to the height of the anchor bean.

RELATED CONCEPTS

“Property Sheets” on page 11

“Beans Palette” on page 9

“Free-Form Surface” on page 8

RELATED TASKS

“Editing Bean Properties” on page 56

“Opening the Property Sheet for a Bean” on page 56

“Chapter 21. Managing the Beans Palette” on page 85

“Undoing and Redoing Changes in the Visual Composition Editor” on page 69

RELATED REFERENCES

“Modify Palette” on page 204

“Chapter 42. Modify Palette Window” on page 209

“Chapter 43. Choose Bean Window” on page 211

“Chapter 53. Resolve Class References” on page 231

“Chapter 52. Morph Into” on page 229

“The Tool Bar in Visual Composition” on page 191

“Chapter 40. Keys” on page 199

Chapter 40. Keys

You can use keys in the following ways while you are using this product:

Use the...	To...
"Window Keys"	Navigate within and among windows.
"Accelerator Keys" on page 200	Speed up certain actions in product windows.
"Help Keys" on page 200	Display help information.

When a plus sign (+) joins two key names, use them together. Hold down the first key and press the second key.

Mnemonics (single underlined characters) are available for menu bar and pull-down choices. To select a menu bar choice using the mnemonics, hold down Alt and enter the mnemonic for the choice that you want. (If the menu bar has the focus, enter only the mnemonic.)

To select a choice on a pull-down menu, enter the mnemonic for the pull-down choice.

Window Keys

Use the...	To...
Alt	Move the focus to and from the menu bar or close the system menu.
Alt+F4	Close the primary window.
Alt+F5	In OS/2, restore the primary window.
Alt+F7	Move the primary window. Use the arrow keys to move the window to a new position and then press Enter.
Alt+F8	Size the primary window. Use the arrow keys to change the size of the window and then press Enter.
Alt+F9	Minimize the primary window to an icon.
Alt+F10	Maximize the primary window.
Alt+Spacebar	Open the system menu for the primary window.
Arrow key	Move the cursor from choice to choice.
Ctrl+Esc	Display the Window List in OS/2 and the Task List in Windows NT.
Ctrl+F5	In Windows platforms, restore the primary window.
Ctrl+F9	In Windows platforms, minimize the primary window to an icon.
Enter	Complete the selection of a menu bar choice or pull-down choice. Also perform the action described on the push button that currently has focus.
Esc	Cancel a pull-down menu or cancel the action window if the window contains a **Cancel** push button.
F10	Move the focus to and from the menu bar or close the system menu.
Page Down or PgDn	Scroll forward a page at a time.
Page Up or PgUp	Scroll backward a page at a time.

Use the...	To...
Spacebar	Select or deselect check boxes and list box choices. Also perform the action described on the button that currently has focus.
Tab	Move the selection cursor from field to field.

Accelerator Keys

You can use the following keys, when applicable, to speed up actions within the product windows:

Use the...	To...
Alt+up arrow	In UNIX platforms, move through all active windows.
Alt+Right Mouse Button	In UNIX platforms, minimize the active window.
Ctrl+Z	Reverse (undo) the operation most recently performed on the bean.
Ctrl+Y	(Redo) Remove the effect of the last **Undo** operation.
Ctrl+Alt+Right Mouse Button	In UNIX platforms, select several parts at a time using the right mouse button.
Ctrl+Click	Select several parts at the same time. In UNIX platforms, use the right mouse button.
Ctrl+Double-click	Assigns the anchor part in multiple selection.
Ctrl+X	Move selected beans or information to the clipboard. **Note:** Select the beans or text before using the Ctrl+Delete keys.
Ctrl+E	Exit (close) the current window.
Ctrl+S	Saves any changes you made to the part.
Ctrl+G	Generate bean code for the bean currently being edited. If the bean has no primary bean, VisualAge saves the bean.
Ctrl+C	Copy selected beans or information to the clipboard.
Ctrl+P	Add a new bean to the beans palette.
Ctrl+V	Load the mouse pointer with the contents of the clipboard.
Delete	Delete the selected beans or text.
Shift+Drag Handles	To size in one direction only.
Shift+Control+Drag	To copy and move the copy.

Help Keys

Use the...	To...
F1	Display general help for the active window. You can use this key on any window.
F11	Display the help index. You can use this key on any window.
Esc	Display the previous help window.
Alt+F4	Close the help window.
Shift+F10	Display help for Help. You can use this on any help window.

Chapter 41. Pop-Up Menus for the Visual Composition Editor

VisualAge provides pop-up menus from the following:

- Free-form surface
- Beans palette
- Beans
- Connections

These menus provide options for adding or making changes to various elements of theVisual Composition Editor or bean design.

Depending on the elements you work with, the following pop-up menu items are available:

- “Add Bean from Project” on page 202
- “Browse Connections” on page 202
- “Change Bean Name” on page 202
- “Change Connection Name” on page 202
- “Change Type” on page 203
- “Connect” on page 203
- “Connectable Features” on page 203
- “Delete” on page 203
- “Event to Code Connection” on page 203
- “Layout” on page 204
- “Modify Palette” on page 204
- “Morph Into” on page 205
- “Open” on page 205
- “Parameter from Code” on page 205
- “Properties” on page 196
- “Promote Bean Feature” on page 205
- “Refresh Palette” on page 205
- “Refresh Interface” on page 205
- “Reorder Connections From” on page 205
- “Restore Shape” on page 205
- “Set Tabbing” on page 206
- “Show Large Icons” on page 206
- “Switch to” on page 206
- “Tear-Off Property” on page 206

Add Bean from Project

Select **Add Bean from Project** from the palette pop-up and the Modify Palette window appears. This version of the Modify Palette window provides an option for you to add beans from your project to the palette.

Browse Connections

Select **Browse Connections** to show or hide connections to or from the bean.

Menu choices

Show To	Displays the connections for which the bean is the target.
Show From	Displays the connections for which the bean is the source.
Show To/From	Displays the connections for which the bean is either the target or source.
Show All	Displays all the connections among beans in the Visual Composition Editor window.
Hide To	Conceals the connections for which the bean is the target.
Hide From	Conceals the connections for which the bean is the source.
Hide To/From	Conceals the connections for which the bean is either the target or source.
Hide All	Conceals all connections among beans in the Visual Composition Editor window.

Change Bean Name

Select **Change Bean Name** to change the name of a bean placed in the Visual Composition Editor.

You can give beans descriptive names to more easily identify them. For example, you can change the default name “Button1” to “Delete.” **Change Bean Name** does not change the label that appears beans such as push buttons.

When you change the name of any bean, you change the beanName property. This name appears in the status area at the bottom of the Visual Composition Editor window as you make connections, and identifies the bean in the beans list.

Note: For subclasses of *java.awt.Component*, the bean name is the same as the name property of the bean.

For a nonvisual bean, the name also appears as text beneath the icon for the bean on the free-form surface.

Change Connection Name

From the connection pop-up, select **Change Connection Name** to change the name of a connection in the Visual Composition Editor.

You can give connections descriptive names to more easily identify them. For example, you can change the default name of a connection that changes the background color from *button1actionPerformed* to ChangeBackgroundColor.

Change Type

Select **Change type** to change the class of a variable bean on the free-form surface. The default class for a dropped variable bean is *java.lang.Object*. The default class for a torn off property variable is the same as the declared type of the property. For example, if you set a panel to CardLayout and tear off its layout property, the variable type is LayoutManager, not CardLayout.

Connect

To make a connection between two beans, select **Connect** from the pop-up menu. When you select **Connect**, the list of preferred features associated with the bean appears.

Select a feature from the **Preferred features** list. If the feature you want is not in the **Preferred features** list, select **Connectable Features**. A connection window appears that lists the available features associated with the bean.

Connectable Features

The **Connectable Features** option is available from the bottom of the **Connect** option on the pop-up. Select **Connectable Features** to see a complete list of the available features (properties, events, and methods) for the bean that you are connecting. The features in the list depend upon the bean and features with which you are working. For example, since an event cannot be a target, the target connections list does not include events.

If the **Connectable Features** pop-up does not display the feature you desire, check the **Show expert features** check box. If the feature is designated **Expert**, it appears in the list.

Delete

Select **Delete** to delete a bean and its connections, or just the bean, or just the connections. You can use multiple select to delete more than one bean at a time.

Event to Code Connection

Select **Event to Code** from the bean pop-up to create a connection that calls a code whenever a specified event occurs. The code can be an existing method or a newly written method. For more information see “Chapter 50. Event-to-Code Connection Window” on page 225.

Layout

Select **Layout** to adjust the placement of beans in a container using null-layout or to adjust the placement of a container bean on the free-form surface. You can adjust the layout of one bean or adjust placement of several beans in a container. If you select one bean, you can adjust its placement horizontally or vertically, by selecting **Distribute**. If you select two beans, you can adjust:

- Alignment to each other—left, center, right, top, middle, bottom
- Relative size to each other—match width, match height, both
- Distribution—horizontally or vertically in the surface

If you select three or more beans, you can make all the above adjustment as well as distribution horizontally or vertically within a bounding box.

Distribute

Select **Distribute** to space visual beans evenly within a specified area.

Horizontally In Bounding Box

Select **Horizontally in bounding box** to space the selected beans evenly between the right and left edges of the bounding box they occupy. This item appears only if three or more beans are selected.

Horizontally In Surface

Use **Horizontally in surface** to space the selected beans evenly between the right and left edges of the bean on which they were dropped. If the selected beans sit directly on the free-form surface, VisualAge distributes them across the entire scrollable width of the free-form surface.

Vertically In Bounding Box

Use **Vertically in bounding box** to space the selected beans evenly between the top and bottom edges of the bounding box they occupy. This item appears only if three or more beans are selected.

Vertically In Surface

Use **Vertically in surface** to space the selected beans evenly between the top and bottom edges of the bean on which they were dropped. If the selected beans sit directly on the free-form surface, VisualAge distributes them across the entire scrollable height of the free-form surface.

Modify Palette

Select **Modify Palette** from the palette pop-up and the Modify Palette window appears.

Morph Into

Use **Morph Into** to change the class or type of a component. For example, you can use this capability in a visual composite to change AWT components to Swing components with few (if any) changes to property or connection settings. For more information before you proceed, see “Chapter 12. Morphing” on page 45.

Open

Select **Open** to open an editor for the bean you selected. If the selected bean has embedded visual beans, the Visual Composition Editor opens for the bean.

Parameter from Code

Select Parameter-from-Code from the bean pop-up to complete a connection that calls code whenever a specified event occurs.

Promote Bean Feature

Use **Promote bean feature** to make a feature of an embedded bean accessible outside the scope of the current composite.

Refresh Palette

Select **Refresh Palette** from the palette pop-up to view changes made to the palette, such as recently added beans and changes to icons. **Refresh Palette** also displays beans for features not loaded at the project level.

Refresh Interface

Select **Refresh Interface** to refresh the bean interface when you add methods or other BeanInfo in the BeanInfo page. The changes are reflected in the Visual Composition Editor.

Reorder Connections From

Select **Reorder Connections From** to change the order in which the connections are executed.

Since the connections from an object with the same notification id run in the order in which they are made, you must use **Reorder Connections From** to place the connections in the order that want them to occur.

Restore Shape

Use **Restore shape** to redraw the selected connections in their original shape.

Set Tabbing

Select **Set Tabbing** to specify the tabbing order for beans that support tabbing. The tabbing order determines the sequence in which beans receive focus when the user presses the Tab, backtab, or cursor movement keys.

The initial tabbing order is determined by the order in which you add the beans. Tabbing options include:

Default Ordering
Sets tabbing order from left to right, top to bottom.

Show Tab Tags
Shows tab tags next to all beans included in the tabbing order.

Hide Tab Tags
Hides the displayed tab tags.

You can change the tab order by dragging the tab tags to the desired order.

Show Large Icons

Select **Show Large Icons** from the palette pop-up to modify the size of icons on the palette and the Beans List. **Show Large Icons** is a toggle with the default set to display 16x16 icon images. The large icons are 32x32.

Switch to

Select **Switch to** when using the CardLayout manager. The CardLayout manager arranges the components in a linear depth sequence (like a deck of cards). **Switch to** enables you to navigate through the deck as follows:

First Arranges the cards so that the first card is on the top of the deck.

Next Moves the next card to the top of the deck.

Previous
Returns the card previously on the top of the deck to the top.

Last Moves the last card in the deck to the top.

Note: Switch to is also available for the Swing bean, JTabbedPane.

Tear-Off Property

Select **Tear-off property** to work with a property as if it were a stand-alone bean. The torn-off property is a variable representing the property and not actually a separate bean.

When you select **Tear-off property**, VisualAge displays the list of properties for the bean you are tearing from. After you select a property from the list, you can drop the torn-off property on the free-form surface. VisualAge creates a connection (represented by a blue double-headed arrow) between the original bean and the torn-off property. You can then form other connections to or from the torn-off property.

RELATED CONCEPTS

“Beans Palette” on page 9

“Chapter 8. Connections” on page 27

“Promotion of Bean Features” on page 23

“Property Sheets” on page 11

“Setting Tabbing Order” on page 12

“Tearing Off Properties” on page 13

“Chapter 12. Morphing” on page 45

RELATED TASKS

“Chapter 19. Connecting Beans” on page 77

“Chapter 21. Managing the Beans Palette” on page 85

“Setting the Tabbing Order” on page 54

“Promoting Bean Features” on page 55

“Editing Bean Properties” on page 56

“Reordering Connections” on page 82

“Tearing Off Properties” on page 56

RELATED REFERENCES

“Chapter 46. Connection Windows” on page 217

“Chapter 50. Event-to-Code Connection Window” on page 225

“Chapter 51. Parameter-from-Code Connection Window” on page 227

“Chapter 44. Promote Features Window” on page 213

“Chapter 42. Modify Palette Window” on page 209

“Chapter 45. Reorder Connections Window” on page 215

Chapter 42. Modify Palette Window

Use this window to perform the following actions:

- Add and remove a category
- Add and remove a bean
- Add and remove a grouping separator
- Reorder beans
- Rename a category

Once you have added beans to the palette, you can place them on the free-form surface, in the beans list, or on an existing container bean, in the same way you place beans that VisualAge provides.

Choose this button...	To perform this action...
Browse	Locate the class/file for the bean.
Add to Category	Add a bean to the selected category.
New Category	Create a new category for the palette.
Rename Category	Change the name of the selected palette category.
Remove	Remove the selected bean or category from the palette.
Add Separator	Place a separator line within the category and drag it to the desired position.
Restore Original Beans	Restore the order and composition of the base categories. User created categories or beans are not affected.
OK	Perform the action and exit the window.
Cancel	Cancel the action.

Bean Type

The bean type field specifies the form of bean that you can add.

Select...	**If you want...**
Class	To add a bean to the palette.
Serialized	To add a serialized bean to the palette.

Class Name or File Name

This field name changes according to the specified bean type.

If you specified...	**The field name is...**
Class	Class Name
Serialized	File Name

If you created the bean, the name you specified appears in the **Name** field.

Open the Modify Palette window from the palette pop-up or by selecting **Bean** and **Modify Palette**. In the **Name** field enter the class name of the bean that you want to add. If you created the bean, this is the same name you specified when you originally created the bean. You can use the **Browse** button to locate the correct File or Class name.

Palette List

The Palette list displays the current categories where you can add a bean. If you create a new category, it appears in this list. You can expand the category for a list of its beans and you can remove beans and categories by selecting the item and then selecting **Remove**.

RELATED CONCEPTS

"Beans Palette" on page 9

RELATED TASKS

"Chapter 21. Managing the Beans Palette" on page 85

"Adding a Bean to the Palette" on page 86

Chapter 43. Choose Bean Window

Select the on the palette to retrieve a bean and drop it on the beans list, free-form surface, or an existing container bean. You must supply the fully qualified class name to add the bean from this window. You can, however, use the **Browse** button to locate the name.

Use the Choose Bean window under the following circumstances:

- When the bean does not appear on the beans palette.
- For beans that you do not use frequently.

Note: You cannot add a bean inside itself and you cannot embed a composite bean inside itself.

Fields

- “Bean Type”
- “Class Name”
- “Name” on page 212

Push buttons

To add the bean or variable to the free-form surface, select **OK**.

Bean Type

You can add a bean as a *class*, a *serialized bean*, or as a *variable*. When you add a bean as a *class*, the default constructor for the class is used when the application runs. This means that a real object is created, not a variable that points to a real object defined elsewhere.

From Bean Type, select the type of bean you want to add.

Select...	**If you want...**
Class	To add an instance of a visual or nonvisual bean
Variable	To add a reference to an instance of a bean
Serialized	To add a serialized bean

Class Name

This field name changes according to the specified bean type.

If you specified...	**The field name is...**
Class	Class Name
Variable	Interface/Class Name
Serialized	File Name

In the **Class name** field, enter the fully qualified name of the Java class. You can use the **Browse** button to locate the name or, if you added the bean previously, it appears in the drop-down list box.

Name

Enter a name in the **Name** field for the bean you want to drop. Bean names may include letters and numbers, but must begin with a letter and include no spaces. This text appears under the bean icon on the free-form surface. If you leave this field blank, VisualAge assigns a **Name** for you.

RELATED CONCEPTS

“Adding Beans in the Visual Composition Editor” on page 10

“Choose Bean Tool” on page 10

RELATED TASKS

“Adding Beans Not on the Palette” on page 68

Chapter 44. Promote Features Window

Use this window to select the methods, properties, and events that you want to add to the public interface for the composite bean.

Fields

- “Method” on page 218
- “Property” on page 218
- “Event” on page 218
- “Promote Name”
- “Details” on page 218
- “Show Expert Features” on page 222

Push buttons

- “>> Promote”
- “<< Remove”

Promote Name

In the **Promoted features** list, double-click the name of the promoted feature and enter the name that you want the property, method, or event to have when added to the public interface for the bean. When you move a feature to the list of promoted features, a default name appears in this field.

>> Promote

Select **>>** to add the property, method, or event to the **Promoted features** list. Select **OK** to add the feature to the public interface of the bean. The feature that you promote appears in the **Connectable Features** window and in the property sheet.

<< Remove

Select **<<** to delete the property, method, or event from the **Promoted features** list. You must first select the feature from the **Promoted features** list.

RELATED CONCEPTS

“Promotion of Bean Features” on page 23

RELATED TASKS

“Promoting Bean Features” on page 55

RELATED REFERENCES

“Promote Bean Feature” on page 205

Chapter 45. Reorder Connections Window

Use this window to change the sequence in which connections from the selected bean are run.

If you make several connections from the same event or property of a bean, the connections for the event or property run in the order in which you made the connections. You can change the sequence by selecting and dragging the listed connection.

RELATED CONCEPTS

“Chapter 8. Connections” on page 27

RELATED TASKS

“Reordering Connections” on page 82

“Chapter 19. Connecting Beans” on page 77

RELATED REFERENCES

“Reorder Connections From” on page 205

Chapter 46. Connection Windows

Use the connection windows to select the feature that you want to use in a connection.

- The Start Connection From window appears when you select **Connectable Features** from the connections pop-up window of the bean you are connecting *from*.
- The Connect *feature_type* Named window appears when you select **Connectable Features** from the preferred features list of the bean you are connecting *to*.

Fields

- “Method” on page 218
- “Property” on page 218
- “Event” on page 218
- “Show Expert Features” on page 222
- “Details” on page 218

The values in these fields vary depending on the following:

- The bean you selected
- If you selected the free-form surface
- How you added features to the bean interface

Push buttons

To use the selected method, property, or event and continue, select **OK**.

The **Set parameters** push button appears when the target of your connection is a method, writable property, or script. Select this push button to specify constant input parameters for the method.

RELATED CONCEPTS

“Chapter 8. Connections” on page 27

RELATED TASKS

“Chapter 19. Connecting Beans” on page 77

“Chapter 20. Manipulating Connections” on page 81

RELATED REFERENCES

“Chapter 47. Property-to-Property Connection Window” on page 219

“Chapter 48. Event-to-Method Connection Window” on page 221

“Chapter 49. Constant Parameter Value Properties Window” on page 223

“Chapter 50. Event-to-Code Connection Window” on page 225

“Chapter 51. Parameter-from-Code Connection Window” on page 227

Method

From the **Method** list, select the method you want to use. The list of methods available depends on the bean you selected. The act of changing or setting the value of a property can be considered a method, so property names might also appear in this list.

Property

From the **Property** list, select the property you want to use. The list of properties available depends on the bean you selected.

Event

From the **Event** list, select the event you want to use. The list of events available depends on the bean you selected.

Details

The **Details** field provides information about the selected feature.

Chapter 47. Property-to-Property Connection Window

Use the Property-to-Property Connection – Properties window to change the source or target of a connection.

Fields

- “Source Property”
- “Pass Event Data” on page 221
- “Target Property”
- “Source Event”
- “Target Event” on page 220
- “Show Expert Features” on page 222

Push buttons

To update the source and target properties or event of the connection and close the window, select **OK**.

To back out of the window without making changes, select **Cancel**.

To reset the source and target connections to the original configuration, select **Reset**.

To switch the source and target properties of the connection, select **Reverse**.

To delete the connection, select **Delete**.

Source Property

The **Source Property** field shows the current source for the connection. To update the connection, select a new source property from the list and then select **OK**.

If you cannot find the property you want, check the **Show expert features** check box. If the property is designated as *expert*, it appears in the list.

Target Property

The **Target Property** field shows the current target for the connection. To update the connection, select a new target property from the list. Then select **OK**.

If you cannot find the property you want, check the **Show expert features** check box. If the property is designated as *expert*, it appears in the list.

Source Event

The **Source Event** field lists the event associated with the source of a connection. To update the connection, select a new source event from the list and then select **OK**.

Setting this field enables you to control whether the data synchronization is unidirectional, bidirectional, or only performed at initialization. This field also enables you to use an unbound property as the source of a connection. When the event is triggered, the target is aligned with the source value. If you do not set this value and the property is not bound, VisualAge allows you to make the connection, but the target property value is not updated when the source property value changes.

Target Event

The **Target Event** field lists the event associated with the target of a connection. To update the connection, select a new target event from the list. Then select **OK**.

Setting this field enables you to control whether the data synchronization is unidirectional, bidirectional, or only performed at initialization. This field also enables you to use an unbound property as the target of property-to-property connections. When the event is triggered, the source is aligned with the target value. If you do not set this value and the property is not bound, VisualAge allows you to make the connection, but the source property value is not updated when the target property value changes.

RELATED CONCEPTS

"Property-to-Property Connections" on page 28

"Chapter 8. Connections" on page 27

RELATED TASKS

"Chapter 19. Connecting Beans" on page 77

"Chapter 20. Manipulating Connections" on page 81

RELATED REFERENCES

"Chapter 46. Connection Windows" on page 217

"Chapter 48. Event-to-Method Connection Window" on page 221

"Chapter 49. Constant Parameter Value Properties Window" on page 223

"Chapter 50. Event-to-Code Connection Window" on page 225

Chapter 48. Event-to-Method Connection Window

Use the Event-to-Method Connection – Properties window to change either end point of an event-to-method connection.

Fields

- "Pass Event Data"
- "Event"
- "Method"
- "Show Expert Features" on page 222

Push buttons

To update the source and target connection features and close the window, select **OK**.

To delete the connection, select **Delete**.

To specify constant parameter values for the target method, select **Set parameters**.

Pass Event Data

The **Pass event data** check box indicates whether connection code will pass data, which is sent in the event notification, to the target as input. The specific nature of the data varies by type of event.

This setting affects the visual cues that VisualAge uses to indicate incomplete connections. Since event data is the first parameter value passed, if the target method or code requires only one parameter and **Pass event data** is checked, the connection appears complete. If the target method or code requires more than one parameter, the connection continues to appear incomplete.

If this box is not checked and inputs are required, VisualAge does not attempt to pass event data to the target, and the connection appears incomplete.

If an event has more than one data parameter and is not specified in another order, the data is passed to the target's parameter in order.

Event

The **Event** field shows the current source event for the connection. To update the connection, select a new source from the **Event** list. Then select **OK**.

Method

The **Method** field shows the current target method for the connection. To update the connection, select a new target from the **Method** list. Then select **OK**.

Show Expert Features

Features that are designated as expert do not appear by default in the feature list. When you select the **Show expert features** check box, VisualAge displays all features, including those designated as expert.

Set Parameters

When you select the **Set parameters** push button, the Constant Parameter Value Properties window opens. Use this window to specify parameter values for the connection.

RELATED CONCEPTS

“Event-to-Method Connections” on page 29

“Chapter 8. Connections” on page 27

“Parameter Connections” on page 30

RELATED TASKS

“Chapter 19. Connecting Beans” on page 77

“Chapter 20. Manipulating Connections” on page 81

“Specifying Expert Features” on page 138

“Specifying Values for Parameters by Default” on page 79

RELATED REFERENCES

“Chapter 46. Connection Windows” on page 217

“Chapter 47. Property-to-Property Connection Window” on page 219

“Chapter 49. Constant Parameter Value Properties Window” on page 223

“Chapter 50. Event-to-Code Connection Window” on page 225

Chapter 49. Constant Parameter Value Properties Window

Use this window to supply a parameter as a constant value. In each parameter field, enter the constant value to be assigned to the specified parameter at run time.

The fields provided in this window depend on the type and number of parameters defined by the method.

RELATED CONCEPTS

“Chapter 8. Connections” on page 27

“Parameter Connections” on page 30

RELATED TASKS

“Editing Connection Properties” on page 80

“Supplying a Parameter Value Using a Constant” on page 79

“Chapter 19. Connecting Beans” on page 77

RELATED REFERENCES

“Chapter 47. Property-to-Property Connection Window” on page 219

“Chapter 48. Event-to-Method Connection Window” on page 221

“Chapter 50. Event-to-Code Connection Window” on page 225

Chapter 50. Event-to-Code Connection Window

Use the Event-to-Code Connection window to create a connection that calls code whenever a specified event occurs.

Fields

- “Method Class”
- “Event”
- “Methods”
- “Code Pane” on page 226
- “Pass Event Data” on page 226

Push buttons

To complete the connection and close the window, select **OK**.

To specify parameter values that are constant, select **Set parameters**.

To close the window without completing the connection, select **Cancel**.

Method Class

The **Method class** field lists the class being edited and all its superclasses. By selecting one of the superclasses, you connect your code to code contained in the superclass. The **Method class** field updates according to the class selected.

Event

From the **Event** list, select the event you want to use. The list of events available depends on the bean you selected.

Methods

This field provides a drop-down list that contains a placeholder name for new methods and the names of methods you previously created. If you select `<new method>` and create a new method, VisualAge assigns a default method name by combining the bean name with the event type. For example, if you create an Event-to-Code Connection with `button1` as the source and *actionPerformed* as the event with no event data passed, VisualAge assigns the name *button1_ActionPerformed* to the new method. You can make the method more descriptive and easier to recognize by changing its name.

Note: The connection name in the beans list is *connEtoC1*.

Code Pane

The method code pane is the large pane located below the event and method fields. Enter your method code in this editable pane. You can also change the name, return value, or parameters of the method by editing the method code. The code pane pop-up menu provides options to assist in editing your code.

Pass Event Data

If you want the event to pass its parameters to the new method, check **Pass event data** at the bottom of the panel.

The **Pass event data** check box indicates whether connection code will pass data, which is sent in the event notification, to the target as input. The specific nature of the data varies by type of event.

This setting affects the visual cues that VisualAge uses to indicate incomplete connections. Since event data is the first parameter value passed, if the target method or code requires only one parameter and **Pass event data** is checked, the connection appears complete. If the target method or code requires more than one parameter, the connection continues to appear incomplete.

If this box is not checked and inputs are required, VisualAge does not attempt to pass event data to the target, and the connection appears incomplete.

If an event has more than one data parameter and is not specified in another order, the data is passed to the target's parameter in order.

If the event and method parameters match in type, VisualAge defaults to **Pass event data**. If the event does not have or does not accept parameters, the default is to not pass event data.

RELATED CONCEPTS

"Code Connections" on page 29

"Chapter 8. Connections" on page 27

RELATED TASKS

"Connecting Features to Code" on page 78

"Supplying a Parameter Value Using a Constant" on page 79

RELATED REFERENCES

"Chapter 46. Connection Windows" on page 217

"Chapter 47. Property-to-Property Connection Window" on page 219

"Chapter 48. Event-to-Method Connection Window" on page 221

"Chapter 49. Constant Parameter Value Properties Window" on page 223

"Chapter 51. Parameter-from-Code Connection Window" on page 227

Chapter 51. Parameter-from-Code Connection Window

Use the Parameter-from-Code Connection window to complete a connection that calls code whenever a specified event occurs. The Parameter-from-Code Connection window is similar to the Event-to-Code Connection window, except it is used to complete an Event-to-Code Connection that requires further parameters.

RELATED CONCEPTS

“Code Connections” on page 29

“Chapter 8. Connections” on page 27

RELATED TASKS

“Connecting Features to Code” on page 78

“Supplying a Parameter Value Using a Constant” on page 79

RELATED REFERENCES

“Chapter 46. Connection Windows” on page 217

“Chapter 47. Property-to-Property Connection Window” on page 219

“Chapter 48. Event-to-Method Connection Window” on page 221

“Chapter 49. Constant Parameter Value Properties Window” on page 223

“Chapter 50. Event-to-Code Connection Window” on page 225

Chapter 52. Morph Into

Use the **Morph Into** window to change the class or type of a component.

To change the class, enter the fully qualified name of the new class in the entry field provided. To pick a class from the list of classes loaded in your workspace, select **Browse**.

To specify a new type, select one of the following. Not all choices may be available, depending on the current class and type of component.

- **Class**, a fully initialized instance.
- **Variable**, an uninitialized instance.
- **Serialized**, a fully initialized, serialized instance. If you select **Serialized** as the new type, you must replace the displayed class name with the name of an .ser file.

Connections to features that are no longer valid in the new class remain until code is regenerated for the composite. To delete such connections instead, select **Delete invalid connections** before you select **OK** to start the process.

RELATED CONCEPTS

"Chapter 12. Morphing" on page 45

"Chapter 2. How Classes and Beans Are Related" on page 3

"Chapter 8. Connections" on page 27

RELATED REFERENCES

"Variable" on page 190

Chapter 53. Resolve Class References

Use the Resolve Class References window to change the class of an unknown component to one that is loaded into your workspace.

When VisualAge encounters an unknown class reference, it attempts to find the correct class name anywhere in the repository. VisualAge then displays the name of the first class name it finds.

Push buttons

To leave the class unresolved for the moment, select **Ignore**.

To pick an alternative class name from the standard class dialog, select **Replace**.

To proceed with the change, select **OK**.

RELATED CONCEPTS

“Chapter 12. Morphing” on page 45

Chapter 54. String Externalization Editor

Use the String Externalization Editor window to specify how you want a given String property value separated for translation. Select one of the following radio buttons:

- **Do not externalize string**, for leaving literal String values in the generated code
- **Externalize string**, for specifying String separation

Resource type radio buttons

- **List resource bundle**
- **Property resource file**

Fields

- **Bundle**, the name of the resource bundle in which to define the String resource
- **Key**, the locale-independent string used to retrieve the resource
- **Value**, the locale-dependent string value of the resource

Push buttons

To create a new resource bundle, select **New**.

To pick from an existing resource bundle, select **Browse**.

RELATED CONCEPTS

"Chapter 14. Internationalization in VisualAge" on page 49

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 25. Separating Strings for Translation" on page 127

Chapter 55. Externalizing: Package.Class

Use this window to specify how you want strings in this class separated for translation.

Resource type radio buttons

- **List resource bundle**
- **Property resource file**

Fields

- An unlabeled entry field through which you specify the resource bundle name. The name of the last bundle accessed, if any, appears in the field. A maximum of eight bundle names is selectable from the drop-down list.
- **Strings to be separated**, which contains the following information. You can edit all values.
  - An unlabeled column that indicates how VisualAge will treat the item. One of the following graphics appear: ✓ **Translate**, ☒ **Never translate**, or ? **Skip**.
  - **Key**, the locale-independent string used to retrieve each resource.
  - **Value**, the locale-dependent string value of a given resource.
- **Context**, which shows you where the selected string occurs.

Push buttons

To create a new resource bundle, select **New**.

To pick from an existing resource bundle, select **Browse**.

This window lists those strings that have not been previously externalized or marked ☒ . To make a string previously marked ☒ appear in this window, find the string in the code and delete the comment at the end of the line: `//$NON-NLS-1$`

RELATED CONCEPTS

"Chapter 14. Internationalization in VisualAge" on page 49

"Chapter 9. Generated Code" on page 33

RELATED TASKS

"Chapter 25. Separating Strings for Translation" on page 127

Chapter 56. BeanInfo Page

Use the BeanInfo page of the Class/Interface Browser to view, define, or modify bean interface features. These features, consisting of properties, events, and methods, represent the characteristics and behavior of your class. When you add features in the BeanInfo page, you define the external view of your bean to consumers who use the bean. By contrast, when you compose a composite bean in the Visual Composition Editor, you define the internal content of the bean.

When you add a new feature in the BeanInfo page, VisualAge generates code for the feature in the bean class. For some features, particularly properties, you might not need to modify this generated code. Additionally, VisualAge generates code that describes the feature in the BeanInfo class for the bean. If a BeanInfo class does not exist for the bean, it is created when you add the first feature in the BeanInfo page. You can also create a BeanInfo class from the **Features** menu. See the JavaSoft Home Page for links to detailed information on JavaBeans and BeanInfo.

If you do not want your bean to inherit features from its superclass, turn off BeanInfo inheritance before a BeanInfo class is created for your bean. If BeanInfo is not inherited from the superclass, only features defined in your bean are available to bean consumers. This means that no inherited features are available for connections or the property sheet when your bean is embedded in another bean. BeanInfo inheritence does not affect accessibility within your class to inherited methods and fields. To control Beaninfo inheritance, open the Options window from the **Window** menu of the Workbench. Then, set the **Inherit BeanInfo of bean superclass** check box in the **Design Time** pane.

If you edit an embedded bean in the Visual Composition Editor and then change its features in the BeanInfo page, be sure to refresh the bean interface when you return to the Visual Composition Editor. Do this by selecting **Refresh Interface** from the bean pop-up menu in the Visual Composition Editor.

From the BeanInfo page, use either the **Features** menu or the tool bar, or both, to manage bean features. Open the **Features** menu either by selecting it from the menu bar **Features** choice or by opening it as a pop-up menu from the **Features** pane.

Fields

- “BeanInfo Page Tool Bar” on page 247
- “Features Pane—BeanInfo Page” on page 238
- “Definitions Pane—BeanInfo Page” on page 239
- “Information Pane—BeanInfo Page” on page 240
- “Source Pane—BeanInfo Page” on page 240
- “Status Area—BeanInfo Page” on page 240

Menu choices

- “Features—BeanInfo Page” on page 241
- “Definitions—BeanInfo Page” on page 243
- “Information—BeanInfo Page” on page 245

RELATED TASKS

“Chapter 27. Defining Bean Interfaces for Visual Composition” on page 131

RELATED REFERENCES

“Chapter 57. BeanInfo Page Menus” on page 241

“Chapter 58. BeanInfo Page Tools” on page 247

Features Pane—BeanInfo Page

Use this pane to view locally defined features for a bean. To view inherited features, browse the bean that defines the features. Initially, all local features except hidden and expert features are listed.

To specify which features are listed, open the **Features** menu, select **Show**, then select a choice to filter the list. The title of the **Features** pane reflects its filtered contents. The features that are listed for each title are as follows:

Features
All features of the bean

Normal Features
All features except hidden and expert features

Method Features
Features that provide bean behavior and access to bean properties

Property Features
Features that represent bean properties

Event Features
Features that report the occurrence of an event in your bean

Expert Features
Features that are marked in BeanInfo as expert

Hidden Features
Features that are marked in BeanInfo as hidden

Each listed feature is preceded by a symbol that indicates the feature type:

P	Property
M	Method
E	Event

The following superscripts provide information about features in the pane:

B	Bound
C	Constrained
E	Expert
H	Hidden
I	Indexed

R	Readable
W	Writeable

If you select a feature in the **Features** pane, the following information appears in other areas of the BeanInfo page:

- Underlying classes, interfaces, and methods of the feature are listed in the **Definitions** pane.
- Bean information for the feature appears in the **Source** pane.
- The fully-qualified name of the feature appears in the status area.

RELATED TASKS

“Specifying Expert Features” on page 138

“Specifying Hidden Features” on page 138

RELATED REFERENCES

“Features—BeanInfo Page” on page 241

“Definitions Pane—BeanInfo Page”

“Information Pane—BeanInfo Page” on page 240

“Source Pane—BeanInfo Page” on page 240

“Status Area—BeanInfo Page” on page 240

“Chapter 56. BeanInfo Page” on page 237

Definitions Pane—BeanInfo Page

Use this pane to list the underlying classes, interfaces, and methods that define features listed in the **Features** pane. If no features are selected, the **Definitions** pane is empty. If you select a feature, definitions for the selected feature are listed.

If you select a definition in the **Definitions** pane, the following information appears in other areas of the BeanInfo page:

- Source code for the class, interface, or method appears in the **Source** pane.
- The fully-qualified name of the class, interface, or method appears in the status area.

RELATED REFERENCES

“Definitions—BeanInfo Page” on page 243

“Features Pane—BeanInfo Page” on page 238

“Information Pane—BeanInfo Page” on page 240

“Source Pane—BeanInfo Page” on page 240

“Status Area—BeanInfo Page” on page 240

“Chapter 56. BeanInfo Page” on page 237

Information Pane—BeanInfo Page

Use this pane to view or edit bean information for a bean or feature. The information is obtained from the BeanInfo class for the bean, if it exists, or by bean reflection.

The title and content of the **Information** pane depend on whether you have selected a feature in the **Features** pane. If a feature is selected, information is displayed for the selected feature. Otherwise, information is displayed for the bean. The title also depends on whether the information is obtained from BeanInfo or by reflection. For example, if information is displayed for a bean by reflection, the **Information** pane title is Bean Reflection Information. If the information is obtained from BeanInfo, the title is Bean Information.

You can modify bean information by selecting an item and editing its value field.

RELATED REFERENCES

"Generated BeanInfo Descriptor Code (an advanced topic)" on page 36

"Information—BeanInfo Page" on page 245

"Features Pane—BeanInfo Page" on page 238

"Chapter 56. BeanInfo Page" on page 237

Source Pane—BeanInfo Page

Use this pane to view or edit source code for a feature definition. The **Source** pane is displayed when you select a class, interface, or method in the **Definitions** pane.

RELATED CONCEPTS

"How Generated Code Coexists with User-Written Code" on page 37

RELATED REFERENCES

"Definitions Pane—BeanInfo Page" on page 239

"Chapter 56. BeanInfo Page" on page 237

Status Area—BeanInfo Page

Use the status area at the bottom of the BeanInfo page to view the fully-qualified name of a selected feature or feature definition.

- If you select a feature in the **Features** pane, the status area shows the feature name.
- If you select a class, interface, or method in the **Definitions** pane, the status area shows the name of the selected item.
- If nothing is selected, a message to that effect is displayed in the status area.

RELATED REFERENCES

"Features Pane—BeanInfo Page" on page 238

"Definitions Pane—BeanInfo Page" on page 239

"Chapter 56. BeanInfo Page" on page 237

Chapter 57. BeanInfo Page Menus

Several menus are available in the BeanInfo page that you can use to work with beans and their features. You can use the following menu bar choices to work with features and related information:

- **Edit**: Use with the **Source** pane
- **Features**: Use with the **Features** pane
- **Definitions**: Use with the **Definitions** pane
- **Information**: Use with the **Information** pane

RELATED REFERENCES

“Features—BeanInfo Page”

“Definitions—BeanInfo Page” on page 243

“Information—BeanInfo Page” on page 245

“Chapter 56. BeanInfo Page” on page 237

Features—BeanInfo Page

Use this menu to define or modify bean interface features. You can open the **Features** menu either as a pull-down menu from the menu bar or as a pop-up menu from the **Features** pane.

You can perform the following tasks from the **Features** menu:

- Create a bean information class for a bean
- List bean interface features for a bean
- Add or remove bean interface features
- Open a feature

Open Opens the Class/Interface Browser for the selected feature

Open to
Opens a menu of bean information choices

Show Opens a menu of filtering choices for the **Features** pane

Sort Opens a menu of ordering choices for the **Features** pane

New BeanInfo Class
Opens a SmartGuide to create a BeanInfo class for the bean

Generate BeanInfo class
Creates a BeanInfo class for the bean

Add Available Features
Opens a dialog to add methods and fields as features

New Property Feature
Opens a SmartGuide to create a new property feature

New Event Set Feature
Opens a SmartGuide to create a new event set feature

New Method Feature
Opens a SmartGuide to create a new method feature

New Listener Interface
Opens a SmartGuide to create a new event listener

Delete Opens a dialog to remove selected features and underlying definitions

RELATED TASKS

"Chapter 27. Defining Bean Interfaces for Visual Composition" on page 131

RELATED REFERENCES

"Features Pane—BeanInfo Page" on page 238

"Show—BeanInfo Page"

"Sort—BeanInfo Page" on page 243

"BeanInfo Class SmartGuide" on page 248

"BeanInfo Class Generator" on page 248

"Add Available Features" on page 254

"New Property Feature SmartGuide" on page 250

"New Event Set Feature SmartGuide" on page 252

"New Method Feature SmartGuide" on page 253

"New Event Listener SmartGuide" on page 251

"Delete Features" on page 254

Show—BeanInfo Page

Use this menu to determine what features are listed in the BeanInfo page **Features** pane. You can open this menu by selecting it from the **Features** menu. Then select a choice to list all features or a subset of features for a bean. The features that are listed for each menu choice are as follows:

Connectable Features
All features of the bean

Normal features
All features except hidden and expert features

Method features
Features that provide bean behavior and access to bean properties

Property features
Features that represent bean properties

Event features
Features that report the occurrence of an event in your bean

Expert features
Features that are marked in BeanInfo as expert

Hidden features
Features that are marked in BeanInfo as hidden

RELATED REFERENCES

"Features Pane—BeanInfo Page" on page 238

"Features—BeanInfo Page" on page 241

Sort—BeanInfo Page

Use this menu to order features in the BeanInfo page **Features** pane. You can open this menu by selecting it from the **Features** menu. Then select a choice to sort features by name or by type.

Sort by type
Orders the features by feature type

Sort by name
Orders the features by name

RELATED REFERENCES

"Features Pane—BeanInfo Page" on page 238

"Features—BeanInfo Page" on page 241

Definitions—BeanInfo Page

Use this menu to work with feature definitions in the **Definitions** pane. You can open the **Definitions** menu either as a pull-down menu from the menu bar or as a pop-up menu from the **Definitions** pane.

Open Opens a browser for the selected method, class, or interface

Open to
Opens a menu of choices for opening another browser:

Project
A project browser for one of the following:

- The class or interface that contains the selected method
- The selected class or interface

Package
A package browser for one of the following:

- The class or interface that contains the selected method
- The selected class or interface

Type A browser for the selected class or interface

References To
Opens a menu of choices to search for the following:

This Method
Calls to the selected method from other methods

Sent Methods
Methods that are called from the selected method

Accessed Fields
Fields that are accessed by the selected method

ReferencedTypes
Classes and interfaces that are referenced by the selected method

This Type
References to the selected class or interface

Field References to a field in the selected class or interface

Static Field
References to a static field in the selected class or interface

Constant
References to a constant in the selected class or interface

Declarations Of
Opens a menu of choices to search for declarations of the following:

This Method
The selected method

Sent Methods
Methods that are called from the selected method

Accessed Fields
fields that are accessed by the selected method

ReferencedTypes
Classes and interfaces that are referenced by the selected method

Replace With
Opens a menu of choices for replacement by one of the following:

Previous Edition
The previous edition of the selected method, class, or interface

Another Edition
Any other edition of the selected method, class, or interface

Manage
Opens a menu of management choices for a class or interface:

Version
Version the open edition of the selected class or interface

Release
Release the open edition of the selected class or interface

Create Open Edition
Create a new open edition of the selected class or interface

Change Owner
Change ownership of the selected class or interface

Compare With
Opens a menu of choices for comparison with one of the following:

Released Edition
The released edition of the selected class or interface

Previous Edition
The previous edition of the selected method, class, or interface

Another Edition
Any other edition of the selected method, class, or interface

Each Other
Selected methods, classes, or interfaces

Document
Opens a menu of choices for replacement by one of the following:

Print Prints source code for the selected class or interface

Generate javadoc
Generates Java API documentation for the selected class or interface

Print Prints source code for the selected method

RELATED REFERENCES

"Definitions Pane—BeanInfo Page" on page 239

Information—BeanInfo Page

Use this menu to work with bean information for a bean or feature in the **Information** pane. You can open the **Information** menu by selecting it from the menu bar.

Revert
Returns the bean information to its previously saved state

Save Saves the current bean information

RELATED REFERENCES

"Information Pane—BeanInfo Page" on page 240

Chapter 58. BeanInfo Page Tools

Several tools are available from the BeanInfo page to help you work with beans and their features. You can access these tools through menus or the tool bar.

"BeanInfo Class Generator" on page 248
Creates or replaces a BeanInfo class for a bean, without user input

"BeanInfo Class SmartGuide" on page 248
Defines information about a bean and creates a BeanInfo class

"Bean Icon Information SmartGuide" on page 249
Specifies icon files for a bean

"Bean Information SmartGuide" on page 249Bean Information
Defines bean information for a new feature

"New Property Feature SmartGuide" on page 250
Defines a new property feature

"New Event Listener SmartGuide" on page 251
Defines a new event listener

"Event Listener Methods SmartGuide" on page 252
Defines methods for a new event listener

"New Event Set Feature SmartGuide" on page 252
Defines a new event set feature

"New Method Feature SmartGuide" on page 253
Defines a new method feature

"Parameter SmartGuide" on page 254
Defines a parameter for a new method feature

"Add Available Features" on page 254
Lists methods that can be added as features

"Delete Features" on page 254
Lists features and underlying methods that can be deleted

"Class Qualification Dialog" on page 254
Selects a fully-qualified class name for a bean

RELATED REFERENCES

"Chapter 56. BeanInfo Page" on page 237

"BeanInfo Page Tool Bar"

BeanInfo Page Tool Bar

Use the tool bar to launch some common tools in the BeanInfo page. The following tools are available from the tool bar:

Open Debugger
Opens a window for debugging

Search
Opens a window to search for a class, interface, constructor, method, or field

New Property Feature
Opens a window to define a new property feature

New Event Set Feature
Opens a window to define a new event set feature

New Method Feature
Opens a window to define a new method feature

RELATED REFERENCES

"New Property Feature SmartGuide" on page 250

"New Event Set Feature SmartGuide" on page 252

"New Method Feature SmartGuide" on page 253

"Chapter 56. BeanInfo Page" on page 237

BeanInfo Class Generator

Use the BeanInfo class generator to create bean information class code for the bean you are working with. This produces bean information for all existing features.

Before you generate the bean information class, bean information for the BeanInfo page is obtained by reflection. After you generate the bean information class, the class is used to find bean information.

RELATED REFERENCES

"Features—BeanInfo Page" on page 241

BeanInfo Class SmartGuide

Use the SmartGuide – BeanInfo Class window to create bean information class code for the bean you are working with. This produces bean information for the bean, but not for existing features. When you add new features, bean information is added to the BeanInfo class.

Before you add any new features, bean information for the BeanInfo page is obtained by reflection. After you generate the bean information class, the class is used to find bean information.

Fields

Display name
The display name represents the BeanInfo class in the VisualAge user interface. This field is optional. If you do not specify a display name, the BeanInfo class name is used.

Short description
The short description is used in the VisualAge user interface. This field is optional. If you do not provide a description, the BeanInfo class name is used.

Customizer
A customizer class provides customized definition of property values for a bean. This field is optional. If you want to provide a custom dialog for

modification of your bean properties, specify a class to support the dialog. If you want to select an existing customizer class, select the **Browse** button.

expert Expert beans do not appear in the Visual Composition Editor by default. However, you can request that expert beans be shown. Mark the bean as expert if you want it to be available in visual composition, but not by default. This option is not initially selected.

hidden
Hidden beans do not appear in some tools, but are available in the Visual Composition Editor. Mark the bean as hidden if you do not want it to be available in other tools. This option is not initially selected.

RELATED REFERENCES

"Class Qualification Dialog" on page 254

"Bean Icon Information SmartGuide"

"Features—BeanInfo Page" on page 241

Bean Icon Information SmartGuide

Use the SmartGuide – Bean Icon Information window to specify the names of files that define icons for your bean. These icons represent the bean on the palette, in the Beans List, and on the free-form surface.

Fields

16X 16 Color
A file that contains a color icon that is 16 pixels wide and 16 pixels high. This field is optional. To specify an icon file, select the **Browse** button.

32X 32 Color
A file that contains a color icon that is 32 pixels wide and 32 pixels high. This field is optional. To specify an icon file, select the **Browse** button.

16X 16 Monochrome
A file that contains a monochrome icon that is 16 pixels wide and 16 pixels high. This field is optional. To specify an icon file, select the **Browse** button.

32X 32 Monochrome
A file that contains a monochrome icon that is 32 pixels wide and 32 pixels high. This field is optional. To specify an icon file, select the **Browse** button.

Bean Information SmartGuide

Use the SmartGuide – Bean Information window to define bean information for a new feature. This information determines how the feature is viewed and accessed in visual composition.

Fields

Display name
The display name represents the feature in the VisualAge user interface. This field is optional. If you do not specify a display name, the feature name is used.

Short description
The short description is used in the VisualAge user interface. This field is optional. If you do not provide a description, the feature name is used.

Property editor
A property editor provides customized definition of a property value. This field is optional. It is available only for property features. If you want to provide a custom dialog to modify a property, specify a property editor class to support the dialog. To select an existing property editor, select the **Browse** button.

For example, you might want to provide a property editor for an *alignment* property that has an integer value. This property can be edited with the registered integer property editor. However, a user can more easily understand descriptive choices, such as *Left*, *Center*, and *Right*. You can create a property editor class that presents these descriptive choices to the user and maps them to integer values for the property.

expert Expert features do not appear in the Visual Composition Editor by default. However, you can request that expert features be shown. Mark the feature as expert if you want it to be available in visual composition, but not by default. This option is not initially selected.

hidden
Hidden features do not appear in the Visual Composition Editor. Mark the feature as hidden if you do not want it to be available in visual composition. This option is not initially selected.

New Property Feature SmartGuide

Use the SmartGuide – New Property Feature window to add a new property feature. Method features are also added to get and set the property. If the property is readable, a get method feature is added, and if the property is writable, a set method feature is added.

If the property is indexed, it contains individually accessible elements. Get and set method features are added to access an element. These are in addition to the get and set method features for the property as a whole. Note that an array property can either be indexed or not. If it is not indexed, array elements are not accessible in the Visual Composition Editor.

If the property is bound, the *propertyChange* event and related method features are also added if they have not yet been added. The *propertyChange* event provides notification of property value changes.

If the property is constrained, the *vetoableChange* event and related method features are also added if they have not yet been added. The *vetoableChange* event provides notification of requested property value changes. If a listener of the *vetoableChange* event throws the *PropertyVetoException*, the property value change is not committed.

Fields

Property name
The name of the property feature. Enter a name for the feature.

Property type
The data type of the property. The type value is initially *java.lang.String*. If

you need a different type, either enter a data type or select the **Browse** button to select a fully-qualified type. You can focus the type search by entering a partial type specification in the **Property type** field before you select the **Browse** button.

Readable
A readable property can report its value. This means that you can make a connection from the property to obtain its value. This option is initially selected.

Writeable
A writable property can have its value modified. This means that you can make a connection to the property to change its value. This option is initially selected.

Indexed
An indexed property contains individually accessible elements. This means that, if the property is readable and writable, you can make a connection to obtain or change the value of an element. This option is initially not selected.

bound
A bound property can report value changes. This means that you can make a connection from the property to obtain the new value whenever it is changed. VisualAge generates code to report the change using the *propertyChange* event. This option is initially selected if a BeanInfo class has been created for the bean.

If a property is not bound, you must associate an event with the source and target properties in connection settings to obtain the value change. VisualAge generates listener methods to get the source property value and set the target property when the event occurs. For example, if you want to obtain the *text* property value from a TextField bean when the Enter key is pressed, you can associate the *actionPerformed* event with the *text* source property and the target property.

constrained
A constrained property can have its value changes vetoed. This means that you can make a connection from the *vetoableChange* event to a method feature or code that could veto the proposed change. This option is initially not selected.

RELATED REFERENCES

“Chapter 9. Generated Code” on page 33

“Class Qualification Dialog” on page 254

“Bean Information SmartGuide” on page 249

“BeanInfo Page Tool Bar” on page 247

“Features—BeanInfo Page” on page 241

New Event Listener SmartGuide

Use the SmartGuide – New Event Listener window to create a new event listener and add it as an event set feature. An *event set* consists of an event listener interface with associated event object and multicaster classes. The multicaster enables multiple listeners for an event.

Method features are also added that other beans can use to add and remove the listener. These methods enable other beans to start and stop listening for the event.

Fields

Event name
The name of the event set feature. Enter a name for the feature.

Event listener
The name of the event listener. This field is initialized when you define the **Event name** field. For example, if the event name is *whatHappened*, the event listener name is initially *WhatHappenedListener*.

Event object
The name of the event object. This field is initialized when you define the **Event name** field. For example, if the event name is *whatHappened*, the event object name is initially *WhatHappenedEvent*.

Event Multicaster
The name of the event multicaster. This field is initialized when you define the **Event name** field. For example, if the event name is *whatHappened*, the event multicaster name is initially *WhatHappenedMulticaster*.

RELATED REFERENCES

"Event Listener Methods SmartGuide"

"Bean Information SmartGuide" on page 249

"Features—BeanInfo Page" on page 241

Event Listener Methods SmartGuide

Use the SmartGuide – New Event Listener window to define one or more methods for a new event listener. These listener methods respond to the event. You must add logic code for each method. The first listener method is added as an event feature. To add additional listener methods as event features, select **Add Available Features** from the **Features** menu. Then, in the Add Available Features window, select the methods you want to add as features.

Fields

Method name
The name of an event method to add to the listener.

Event Listener methods
A list of methods for the event listener. To add the method in the **Method name** field as a listener method, select the **Add** button. To remove the selected method from the list, select the **Remove** button.

New Event Set Feature SmartGuide

Use the SmartGuide – New Event Set Feature window to select an existing event listener and add it as an event set feature. An *event set* consists of an event listener interface with associated event object and multicaster classes. The multicaster enables multiple listeners for an event.

The event listener contains one or more methods that respond to the event. Each listener method is added as an event feature. Method features are also added that other beans can use to add or remove the listener. These methods enable other beans to start and stop listening for the event.

Fields

Event name

The name of the event set feature. The initial selection is *action*. You can select a different event set from the drop-down list.

Event listener

The name of the event listener. The initial selection depends on the selected event set. For the *action* event, the initial listener is *java.awt.event.ActionListener*. You can select a different event listener from the drop-down list. If you need to select a fully-qualified name for the particular listener you want, select the **Browse** button for the listener.

RELATED REFERENCES

“Class Qualification Dialog” on page 254

“Bean Information SmartGuide” on page 249

“BeanInfo Page Tool Bar” on page 247

“Features—BeanInfo Page” on page 241

New Method Feature SmartGuide

Use the SmartGuide – New Method Feature window to add a new method feature. The SmartGuide creates a new public class method for the feature. You must add logic code for the method.

Fields

Method name

The name of the method feature. Enter a name for the feature.

Return type

The return data type of the method feature. The type value is initially *void*. If you need a different type, either enter a data type or select the **Browse** button to select a fully-qualified type. You can focus the type search by entering a partial type specification in the **Return type** field before you select the **Browse** button.

Parameter count

The number of parameters for the method feature. The initial selection is *0*. You can select a different number from the drop-down list. You define each parameter with the parameter SmartGuide.

RELATED CONCEPTS

“How Generated Code Coexists with User-Written Code” on page 37

RELATED REFERENCES

“Class Qualification Dialog” on page 254

“Parameter SmartGuide” on page 254

“Bean Information SmartGuide” on page 249

“BeanInfo Page Tool Bar” on page 247

“Features—BeanInfo Page” on page 241

Parameter SmartGuide

Use the SmartGuide – Parameter window to define a parameter for a new method feature.

Fields

Parameter name
The name of the method parameter. Enter a name for the parameter.

Parameter type
The data type of the parameter. The type value is initially *boolean*. If you need a different type, either enter a data type or select the **Browse** button to select a fully-qualified type. You can focus the type search by entering a partial type specification in the **Return type** field before you select the **Browse** button.

Display name
The display name is used in the VisualAge user interface. This field is optional. If you do not specify a display name, the parameter name is used.

Short description
The short description is used in the VisualAge user interface. This field is optional. If you do not provide a description, the parameter name is used.

RELATED REFERENCES

"Class Qualification Dialog"

Add Available Features

Use the Add Available Features window to add features based on public methods that are not defined as features. VisualAge finds all available public methods of the class that have not been added as features, and lists them for you to select. For example, methods that you add in the Methods page can be added as features. Get and set methods for fields can be added as property features.

Delete Features

Use the Delete features window to remove features and underlying methods. Features that you selected for deletion are listed in the **All the following feature(s) will be deleted:** field. The methods that define the features are listed in the **The following selected method(s) will be deleted:** field. If you want to delete any of these methods, select them. Select **OK** to delete features and methods.

Class Qualification Dialog

Use the Class Qualification Dialog to select a fully-qualified class or interface name for a field in another window. The initial scope of the class or interface search is determined by input to this dialog when it is opened.

Fields

Pattern

A selection pattern that determines class and interface names listed in the **Class/Interface Names** field. The initial pattern depends on information passed to this dialog.

Class/Interface Names

A list of unqualified class and interface names that match the selection pattern. If there is no selection pattern, all available classes and interfaces are listed.

Package Names

A list of packages that contain the selected class or interface. Select the package that contains the class or interface you want. The selected package name is used to qualify the selected class or interface name returned to the calling window.