

# Designing Applications with JBuilder®

## JBuilder® 2005

**Borland®**  
Excellence Endures™

Borland Software Corporation  
100 Enterprise Way  
Scotts Valley, California 95066-3249  
[www.borland.com](http://www.borland.com)

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997–2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JB2005designui 6E5R0804  
0405060708-9 8 7 6 5 4 3 2 1  
PDF

# Contents

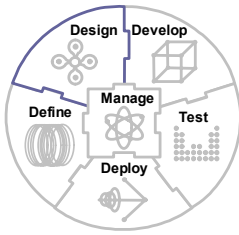
|                                                       |           |  |
|-------------------------------------------------------|-----------|--|
| <b>Chapter 1</b>                                      |           |  |
| <b>Visual design in JBuilder</b>                      | <b>1</b>  |  |
| Requirements for a class to be                        |           |  |
| visually designable . . . . .                         | 2         |  |
| Starting with wizards . . . . .                       | 3         |  |
| Understanding JavaBeans . . . . .                     | 3         |  |
| Understanding containers . . . . .                    | 4         |  |
| Types of containers . . . . .                         | 4         |  |
| Understanding component libraries . . . . .           | 4         |  |
| Documentation conventions . . . . .                   | 5         |  |
| Developer support and resources . . . . .             | 6         |  |
| Contacting Borland Developer Support . . . . .        | 6         |  |
| Online resources . . . . .                            | 7         |  |
| World Wide Web . . . . .                              | 7         |  |
| Borland newsgroups . . . . .                          | 7         |  |
| Usenet newsgroups . . . . .                           | 7         |  |
| Reporting bugs . . . . .                              | 8         |  |
| <b>Chapter 2</b>                                      |           |  |
| <b>Creating JavaBeans</b>                             |           |  |
| <b>with BeansExpress</b>                              | <b>9</b>  |  |
| What is a JavaBean? . . . . .                         | 9         |  |
| Why build JavaBeans? . . . . .                        | 9         |  |
| Generating a bean class . . . . .                     | 10        |  |
| Designing the user interface of your bean . . . . .   | 11        |  |
| Adding properties to your bean . . . . .              | 11        |  |
| Modifying a property . . . . .                        | 13        |  |
| Removing a property . . . . .                         | 14        |  |
| Adding bound and constrained properties . . . . .     | 14        |  |
| Creating a BeanInfo class . . . . .                   | 15        |  |
| Specifying BeanInfo data for a property . . . . .     | 15        |  |
| Working with the BeanInfo designer . . . . .          | 16        |  |
| Modifying a BeanInfo class . . . . .                  | 16        |  |
| Adding events to your bean . . . . .                  | 17        |  |
| Firing events . . . . .                               | 17        |  |
| Listening for events . . . . .                        | 19        |  |
| Creating a custom event set . . . . .                 | 20        |  |
| Creating a property editor . . . . .                  | 21        |  |
| Creating a String List editor . . . . .               | 22        |  |
| Creating a String Tag List editor . . . . .           | 22        |  |
| Creating an Integer Tag List editor . . . . .         | 23        |  |
| Creating a custom component property editor . . . . . | 24        |  |
| Adding support for serialization . . . . .            | 25        |  |
| Checking the validity of a JavaBean . . . . .         | 25        |  |
| Installing a bean on the component palette . . . . .  | 26        |  |
| <b>Chapter 3</b>                                      |           |  |
| <b>Introducing the visual designer</b>                | <b>27</b> |  |
| Using the designer . . . . .                          | 27        |  |
| The design surface . . . . .                          | 28        |  |
| The component palette . . . . .                       | 29        |  |
| Using the Selection Arrow . . . . .                   | 30        |  |
| Using the Bean Chooser . . . . .                      | 30        |  |
| The Inspector . . . . .                               | 31        |  |
| The component tree . . . . .                          | 31        |  |
| Designer categories . . . . .                         | 31        |  |
| UI designer . . . . .                                 | 32        |  |
| Menu designer . . . . .                               | 32        |  |
| Data Access designer . . . . .                        | 32        |  |
| Default designer . . . . .                            | 32        |  |
| Keyboarding in the designer . . . . .                 | 33        |  |
| <b>Chapter 4</b>                                      |           |  |
| <b>Using the component tree</b>                       |           |  |
| <b>and Inspector</b>                                  | <b>35</b> |  |
| Using the component tree . . . . .                    | 35        |  |
| Opening particular designer types . . . . .           | 36        |  |
| Adding components . . . . .                           | 36        |  |
| Cutting, copying, and pasting components . . . . .    | 37        |  |
| Deleting components . . . . .                         | 38        |  |
| Using Undo and Redo . . . . .                         | 38        |  |
| Changing a component name . . . . .                   | 38        |  |
| Moving components . . . . .                           | 38        |  |
| Viewing component class names . . . . .               | 39        |  |
| Understanding component tree icons . . . . .          | 39        |  |
| Using the Inspector . . . . .                         | 39        |  |
| Surfacing property values . . . . .                   | 40        |  |
| Making properties class variables . . . . .           | 40        |  |
| Setting property exposure . . . . .                   | 41        |  |
| Setting property values . . . . .                     | 41        |  |
| Setting shared properties . . . . .                   | 42        |  |
| Setting a property when the                           |           |  |
| drop-down list is empty . . . . .                     | 42        |  |
| Understanding the Inspector . . . . .                 | 42        |  |
| <b>Chapter 5</b>                                      |           |  |
| <b>Handling events</b>                                | <b>45</b> |  |
| Attaching event-handling code . . . . .               | 46        |  |
| Creating a default event handler . . . . .            | 46        |  |
| Deleting event handlers . . . . .                     | 46        |  |
| Connecting controls and events . . . . .              | 47        |  |
| Standard event adapters . . . . .                     | 48        |  |
| Anonymous inner class adapters . . . . .              | 48        |  |
| Choosing event handler style . . . . .                | 49        |  |
| Examples: connecting and handling events . . . . .    | 49        |  |
| Displaying text when a button is pressed . . . . .    | 49        |  |
| <b>Chapter 6</b>                                      |           |  |
| <b>Creating user interfaces</b>                       | <b>53</b> |  |
| Selecting components in the UI . . . . .              | 54        |  |
| Adding to nested containers . . . . .                 | 54        |  |
| Moving and resizing components . . . . .              | 55        |  |
| Managing the design . . . . .                         | 56        |  |
| Grouping components . . . . .                         | 56        |  |
| Adding application-building components . . . . .      | 56        |  |
| Menus . . . . .                                       | 57        |  |
| Dialog boxes . . . . .                                | 57        |  |
| Database components . . . . .                         | 59        |  |

|                                                  |           |                                                     |     |
|--------------------------------------------------|-----------|-----------------------------------------------------|-----|
| Changing look and feel . . . . .                 | 59        | Understanding sizing properties. . . . .            | 85  |
| Runtime look and feel . . . . .                  | 59        | Determining the size and location of                |     |
| Design time look and feel . . . . .              | 61        | your UI window at runtime . . . . .                 | 86  |
| Testing the UI at runtime . . . . .              | 61        | Sizing a window automatically with pack() . . . . . | 86  |
| <b>Chapter 7</b>                                 |           | Calculating preferredSize for containers . . . . .  | 86  |
| <b>Designing menus</b>                           | <b>63</b> | Portable layouts. . . . .                           | 86  |
| Opening the Menu designer . . . . .              | 63        | XYLayout . . . . .                                  | 87  |
| Menu terminology. . . . .                        | 64        | Explicitly setting the size of a window             |     |
| Menu design tools . . . . .                      | 64        | using setSize() . . . . .                           | 87  |
| Creating menus. . . . .                          | 65        | Making the size of your UI portable                 |     |
| Adding menu items . . . . .                      | 66        | to various platforms. . . . .                       | 87  |
| Inserting and deleting menus                     |           | Positioning a window on the screen . . . . .        | 87  |
| and menu items . . . . .                         | 66        | Placing the sizing and positioning                  |     |
| Inserting separators . . . . .                   | 67        | method calls in your code . . . . .                 | 88  |
| Specifying accelerator keys . . . . .            | 67        | Adding custom layout managers . . . . .             | 88  |
| Disabling (dimming) menu items . . . . .         | 67        | Layouts provided with JBuilder . . . . .            | 89  |
| Creating checkable menu items . . . . .          | 67        | null. . . . .                                       | 90  |
| Creating Swing radio button menu items . . . . . | 68        | XYLayout . . . . .                                  | 90  |
| Moving menu items. . . . .                       | 69        | Aligning components in XYLayout . . . . .           | 91  |
| Creating submenus. . . . .                       | 69        | Alignment options for XYLayout . . . . .            | 91  |
| Moving existing menus to submenus . . . . .      | 70        | BorderLayout . . . . .                              | 92  |
| Attaching code to menu events. . . . .           | 70        | Setting constraints . . . . .                       | 93  |
| Creating pop-up menus. . . . .                   | 71        | FlowLayout . . . . .                                | 94  |
|                                                  |           | Alignment . . . . .                                 | 94  |
|                                                  |           | Gap. . . . .                                        | 94  |
|                                                  |           | Order of components. . . . .                        | 94  |
| <b>Chapter 8</b>                                 |           | VerticalFlowLayout . . . . .                        | 95  |
| <b>Advanced topics</b>                           | <b>73</b> | Alignment . . . . .                                 | 95  |
| Managing the component palette. . . . .          | 73        | Gap. . . . .                                        | 95  |
| Adding a component to the component              |           | Horizontal fill . . . . .                           | 96  |
| palette . . . . .                                | 74        | Vertical fill . . . . .                             | 96  |
| Selecting an image for a component               |           | Order of components. . . . .                        | 96  |
| palette button . . . . .                         | 75        | BoxLayout2 . . . . .                                | 97  |
| Adding pages to the component palette. . . . .   | 75        | GridLayout. . . . .                                 | 97  |
| Removing a page or component from                |           | Columns and rows . . . . .                          | 97  |
| the component palette . . . . .                  | 76        | Gap. . . . .                                        | 97  |
| Reorganizing the component palette . . . . .     | 76        | CardLayout . . . . .                                | 98  |
| Handling red beans. . . . .                      | 76        | Creating a CardLayout container. . . . .            | 98  |
| Serializing . . . . .                            | 77        | Creating the controls. . . . .                      | 99  |
| Serializing components in JBuilder . . . . .     | 77        | Specifying the gap . . . . .                        | 99  |
| Serializing a this object . . . . .              | 77        | OverlayLayout2 . . . . .                            | 100 |
| Using customizers in the designer . . . . .      | 79        | GridBagLayout . . . . .                             | 100 |
| Modifying beans with customizers. . . . .        | 79        | Display area . . . . .                              | 101 |
| Handling resource bundle strings. . . . .        | 79        | About GridBagConstraints . . . . .                  | 102 |
| Removing from resource bundles . . . . .         | 79        | Setting GridBagConstraints manually                 |     |
| Adding to resource bundles . . . . .             | 80        | in the source code . . . . .                        | 103 |
| <b>Chapter 9</b>                                 |           | Modifying existing GridBagLayout code               |     |
| <b>Using layout managers</b>                     | <b>81</b> | to work in the designer . . . . .                   | 103 |
| About layout managers . . . . .                  | 81        | Designing GridBagLayout visually in                 |     |
| Using null and XYLayout . . . . .                | 82        | the designer . . . . .                              | 103 |
| Understanding layout properties . . . . .        | 83        | Converting to GridBagLayout . . . . .               | 104 |
| Understanding layout constraints . . . . .       | 83        | Adding components to a GridBagLayout                |     |
| Examples of layout properties                    |           | container . . . . .                                 | 105 |
| and constraints. . . . .                         | 83        | Setting GridBagConstraints in the                   |     |
| Using container layouts. . . . .                 | 84        | GridBagConstraints Editor . . . . .                 | 105 |
| Modifying layout properties . . . . .            | 84        | Displaying the grid . . . . .                       | 106 |
| Modifying component layout constraints . . . . . | 85        | Using the mouse to change constraints . . . . .     | 106 |
|                                                  |           | Using the GridBagLayout context menu . . . . .      | 106 |

|                                                  |     |                                                |            |
|--------------------------------------------------|-----|------------------------------------------------|------------|
| GridBagConstraints . . . . .                     | 107 | Prototyping your UI . . . . .                  | 119        |
| anchor . . . . .                                 | 107 | Use null or XYLayout for prototyping . . . . . | 119        |
| fill . . . . .                                   | 108 | Design the big regions first . . . . .         | 119        |
| gridwidth, gridheight . . . . .                  | 108 | Save before experimenting . . . . .            | 119        |
| gridx, gridy . . . . .                           | 109 | Using nested panels and layouts . . . . .      | 120        |
| insets . . . . .                                 | 109 |                                                |            |
| ipadx, ipady . . . . .                           | 110 | <b>Appendix A</b>                              |            |
| weightx, weighty . . . . .                       | 111 | <b>Migrating files from other Java IDEs</b>    | <b>121</b> |
| Sample GridBagConstraints source code . . . . .  | 114 | VisualAge . . . . .                            | 121        |
| PaneLayout . . . . .                             | 116 | Forte . . . . .                                | 121        |
| PaneConstraints variables . . . . .              | 116 | VisualCafé . . . . .                           | 122        |
| How components are added to PaneLayout . . . . . | 117 |                                                |            |
| Creating a PaneLayout container                  |     | <b>Index</b>                                   | <b>123</b> |
| in the designer . . . . .                        | 117 |                                                |            |
| Modifying the component location                 |     |                                                |            |
| and size in the Inspector . . . . .              | 118 |                                                |            |

# Figures

|     |                                                |    |      |                                |     |
|-----|------------------------------------------------|----|------|--------------------------------|-----|
| 3.1 | The designer . . . . .                         | 28 | 9.6  | verticalFill example . . . . . | .96 |
| 9.1 | XYLayout example . . . . .                     | 90 | 9.7  | GridLayout example . . . . .   | .97 |
| 9.2 | BorderLayout's placement constraints . . . . . | 92 | 9.8  | CardLayout example . . . . .   | .98 |
| 9.3 | FlowLayout example . . . . .                   | 94 | 9.9  | GridBagLayout example. . . . . | 100 |
| 9.4 | VerticalFlowLayout example . . . . .           | 95 | 9.10 | PanelLayout example . . . . .  | 116 |
| 9.5 | horizontalFill example . . . . .               | 96 |      |                                |     |



# Visual design in JBuilder

JBuilder's designer allows you to create and change visually designable files quickly and efficiently. This documentation explores the four visible aspects of the designer — the component tree, the design surface, the Inspector, and the component palette — and walks you through designing a user interface and creating menus. Tutorials follow that explore design, event-handling, and layout managers in greater detail.

## See also

- “Using the JBuilder workspace” in *Getting Started with JBuilder* to review the layout and terminology of the IDE

*Designing Applications with JBuilder* contains the following chapters:

- [Chapter 3, “Introducing the visual designer”](#)

Provides an overview of the visual design tools in JBuilder. Names the different parts of the designer and describes what each part does and how it relates to the others. Explains the different types of designer, such as the UI designer and the Menu designer.

- [Chapter 4, “Using the component tree and Inspector”](#)

Explains how to use the component tree and the Inspector. Describes how they work together to handle and edit components.

- [Chapter 5, “Handling events”](#)

Describes how to add and delete event handlers for components using the Inspector. Gives specific examples of how to code commonly used event handlers for the JBuilder dialog components.

- [Chapter 2, “Creating JavaBeans with BeansExpress”](#)

Describes how to create a JavaBean and how to convert an existing class to a JavaBean.

- [Chapter 6, “Creating user interfaces”](#)

Explains how to design a user interface using JBuilder's visual design tools. Also explains how to attach code to a component's event handlers, and gives specific examples of how to hook up common events to UI elements such as menus and toolbar buttons.

- [Chapter 7, “Designing menus”](#)  
Explains how to create menus using JBuilder’s Menu designer.
- [Chapter 8, “Advanced topics”](#)  
Provides information on advanced topics and topics pertinent to distributed application development. Explains how to serialize in JBuilder, create customizers, and handle resource bundles.
- [Chapter 9, “Using layout managers”](#)  
Explains Java layout managers and describes how to work with each of the layout managers within the UI designer.
- [Appendix A, “Migrating files from other Java IDEs”](#)  
Explains how to handle code developed in other Java IDEs so the file can be visually designed using JBuilder’s visual design tools.

## Requirements for a class to be visually designable

---

For a file to be visually designable, it must

- Be a `.java` file.
- Be free from syntax errors.
- Use a default public constructor.
- Define a class whose name matches the file name.
  - The defined class must not be an inner or anonymous class.
  - The defined class must not be an interface.

Any file that meets the above requirements can be designed using the component tree and the Inspector. This allows you to visually design both UI and non-UI classes.

**Note** JavaBeans meet these requirements. These requirements are also met when you create your files with JBuilder’s JavaBean, Application, Applet, Frame, Panel, and Dialog wizards.

When you first add a component to your design, the JBuilder visual design tools will make sure that

- The class has a default public constructor.
- The class has a `jbInit()` method.
  - This `jbInit()` method is called correctly from the default constructor.

**Note** If the class does not have a default public constructor, the component appears as a red square in the Design view.

**Important** If you are migrating files from other Java IDEs into JBuilder, you might need to modify your code so JBuilder’s designers can work with the files. JBuilder’s visual design tools can recognize VisualAge files as long as they meet the requirements of a visually designable file.

### See also

- [Appendix A, “Migrating files from other Java IDEs”](#)
- [“Handling red beans” on page 76](#)



## Starting with wizards

---

The first step in designing a user interface with JBuilder is to create or open a designable container class, such as a `Frame` or a `Panel`.

Choose `File|New` to open the object gallery. Several pages of the object gallery provide access to wizards that generate visually designable files, including `Applet`, `Application`, and `Dialog`.

The wizards import all necessary packages. Just open the container file, click the `Design` tab in the content pane, and start using the designer.

**Note** You can add additional frames, panels, and dialog boxes to your project by choosing `File|New` and selecting the appropriate wizard from the object gallery.

## Understanding JavaBeans

---

JavaBeans are self-contained classes that don't need any other classes to complete them, but are designed to be customized and to communicate effectively with others so they can work together gracefully. Think of a metaphor: a wheel is designed to rotate around a central axle, and can do that without any further tooling. Wheels can be customized to fit under cars or inside pulleys, where they interface with the other components of the design to perform a larger function.

JavaBeans must support these features:

*Introspection* Lets beans be analyzed.

*Customization* Lets the appearance and behavior of beans be tailored as needed.

*Events* Allow beans to communicate.

*Persistence* Lets a bean's runtime state be saved.

A `JavaBean` may also have a `BeanInfo` class. `BeanInfo` describes its component to the design tools clearly and effectively. JBuilder looks first for `BeanInfo` for a bean, and when it doesn't find it, it uses introspection to discover the bean's properties and attributes.

JavaBeans describe *components*. Components are the building blocks used by JBuilder's visual design tools to build a program. You build your program by choosing, customizing, and connecting components. JBuilder comes with a set of ready-to-use components on the component palette. Supplement these by creating new components or by installing third-party components.

Examine JavaBeans using `BeanInsight`. Select `Tools|BeanInsight` and type in the name of any compiled bean to examine its properties, events, customizers, and so on.

### See also

- The JavaBeans specification at <http://www.javasoft.com/beans/docs/spec.html>
- [“Managing the component palette” on page 73](#)

## Understanding containers

---

*Containers* are a special type of component that hold and manage other components. Containers extend `java.awt.Container`. Containers generally appear as panels, frames, and dialog boxes in a running program. Generally, your visible design work in JBuilder takes place in containers.

### Types of containers

Containers can be broadly categorized into windows and panels.

- A *window* is a standalone, top-level container component without borders, title bar, or menu bar. Although a window can be used to implement a pop-up window, such as a splash screen, it's more common to use a subclass of `java.awt.Window` in your UI, such as one of those listed below, rather than the actual `Window` class.
  - *Frame*  
A top-level window with a border and a title. A frame has standard window controls such as a control menu, buttons to minimize and maximize the window, and controls to resize the window. It can also contain a menu bar.  
  
Typically, the main UI container for a Java application (as opposed to an applet) is a customized subclass of `java.awt.Frame`. The customization typically instantiates and positions other components on the frame, sets labels, attaches controls to data, and so forth.
  - *Dialog*  
A pop-up window, similar to a frame, but it can't contain a menu bar. A dialog is used for getting input and giving warnings, and is usually intended to be temporary. It can be one of the following types:
    - *Modal*: Prevents input to any other windows in the application until that dialog is dismissed.
    - *Modeless*: Allows information to be entered in both the dialog and the application.
- A *panel* is a simple UI container, without border or caption, used to group other components, such as buttons, check boxes, and text fields. A panel is embedded within some other UI component, such as `Frame` or `Dialog`. It can also be nested within other panels.
- *Applet*  
A subclass of `Panel` used to build a program intended to be embedded in an HTML page and run in an HTML browser or applet viewer. Since `Applet` is a subclass of `Panel`, it can contain components but does not have a border or caption.

## Understanding component libraries

---

Component libraries are collections of ready-made components.

JBuilder supplies a number of libraries of JavaBean components on the component palette, including Swing, `dbSwing`, XML, EJB, AWT, and others. Some, such as Swing and AWT, consist only of industry-standard components. Others provide proprietary components which JBuilder provides for your convenience.

Components in a single library generally share an important high-level commonality. Where component features overlap within or between libraries, differences in behavior, look and feel, or requirements of the components can help you decide which component to choose.

For instance, AWT components have the advantage of being compatible with both older and newer versions of internet browsers. They have the disadvantage of being heavyweight, porting across platforms rather poorly, and not being readily customizable. Swing components have the advantages of being lightweight, designed to look the same across platforms, and providing customization such as look-and-feel and painting options. The `dbSwing` library consists of subclasses of Swing components that have the added advantage of a `dataSet` property and a `columnName` property to make the `Swing` components data-aware.

By comparing components that perform related tasks, you can determine which components are best suited to a particular task. In many cases, several components can perform the desired action, but you might choose one based on how it looks and works, how easy it is to use, or whether it makes sense for the tools you expect your users to have.

## Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

**Table 1.1** Typeface and symbol conventions

| Typeface       | Meaning                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Bold</b>    | Bold is used for java tools, <code>bmj</code> (Borland Make for Java), <code>bcj</code> (Borland Compiler for Java), and compiler options. For example: <b><code>javac</code></b> , <b><code>bmj</code></b> , <b><code>-classpath</code></b> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <i>Italics</i> | Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <i>Keycaps</i> | This typeface indicates a key on your keyboard, such as "Press <i>Esc</i> to exit a menu."                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Monospace type | Monospaced type represents the following: <ul style="list-style-type: none"> <li>■ text as it appears onscreen</li> <li>■ anything you must type, such as "Type <code>Hello World</code> in the Title field of the Application wizard."</li> <li>■ file names</li> <li>■ path names</li> <li>■ directory and folder names</li> <li>■ commands, such as <code>SET PATH</code></li> <li>■ Java code</li> <li>■ Java data types, such as <code>boolean</code>, <code>int</code>, and <code>long</code>.</li> <li>■ Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events</li> <li>■ argument names</li> <li>■ field names</li> <li>■ Java keywords, such as <code>void</code> and <code>static</code></li> </ul>                                                                                                                                                                                                                                                      |
| [ ]            | Square brackets in text or syntax listings enclose optional items. Do not type the brackets.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| < >            | <p>Angle brackets are used to indicate variables in directory paths, command options, and code samples.</p> <p>For example, <code>&lt;filename&gt;</code> may be used to indicate where you need to supply a file name (including file extension), and <code>&lt;username&gt;</code> typically indicates that you must provide your user name.</p> <p>When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (<code>&lt; &gt;</code>). For example, you would replace <code>&lt;filename&gt;</code> with the name of a file, such as <code>employee.jds</code>, and omit the angle brackets.</p> <p><b>Note:</b> Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <code>&lt;font color=red&gt;</code> and <code>&lt;ejb-jar&gt;</code>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.</p> |

**Table 1.1** Typeface and symbol conventions (continued)

| Typeface              | Meaning                                                                                                                                                                                                           |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Italics, serif</i> | This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, <code>&lt;url="jdbc:borland:jbuilder\samples\guestbook.jds"&gt;</code> |
| ...                   | In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.  |

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

**Table 1.2** Platform conventions

| Item           | Meaning                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Paths          | Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Home directory | The location of the standard home directory varies by platform and is indicated with a variable, <code>&lt;home&gt;</code> . <ul style="list-style-type: none"> <li>■ For UNIX and Linux, the home directory can vary. For example, it could be <code>/user/&lt;username&gt;</code> or <code>/home/&lt;username&gt;</code></li> <li>■ For Windows NT, the home directory is <code>C:\Winnt\Profiles\&lt;username&gt;</code></li> <li>■ For Windows 2000 and XP, the home directory is <code>C:\Documents and Settings\&lt;username&gt;</code></li> </ul> |
| Screen shots   | Screen shots reflect the Borland Look & Feel on various platforms.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

### Contacting Borland Developer Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support upon installation of the Borland product, to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at <http://www.borland.com/devsupport/>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

## Online resources

---

You can get information from any of these online sources:

### World Wide Web

<http://www.borland.com/>  
<http://info.borland.com/techpubs/jbuilder/>

**Electronic newsletters** To subscribe to electronic newsletters, use the online form at:  
<http://www.borland.com/products/newsletters/index.html>

## World Wide Web

---

Check the JBuilder page of the Borland website, [www.borland.com/jbuilder](http://www.borland.com/jbuilder), regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- <http://www.borland.com/jbuilder/> (updated software and other files)
- <http://info.borland.com/techpubs/jbuilder/> (updated documentation and other files)
- <http://bdn.borland.com/> (contains our web-based news magazine for developers)

## Borland newsgroups

---

When you register JBuilder you can participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at <http://www.borland.com/newsgroups/>.

## Usenet newsgroups

---

The following Usenet groups are devoted to Java and related programming issues:

- [news:comp.lang.java.advocacy](#)
- [news:comp.lang.java.announce](#)
- [news:comp.lang.java.beans](#)
- [news:comp.lang.java.databases](#)
- [news:comp.lang.java.gui](#)
- [news:comp.lang.java.help](#)
- [news:comp.lang.java.machine](#)
- [news:comp.lang.java.programmer](#)
- [news:comp.lang.java.security](#)
- [news:comp.lang.java.softwaretools](#)

**Note** These newsgroups are maintained by users and are not official Borland sites.

## Reporting bugs

---

If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

- **Support Programs page** at <http://www.borland.com/devsupport/namerica/>. Click the Information link under “Reporting Defects” to open the Welcome page of Quality Central, Borland’s bug-tracking tool.
- **Quality Central** at <http://qc.borland.com>. Follow the instructions on the Quality Central page in the “Bugs Report” section.
- **Quality Central menu command** on the main Tools menu of JBuilder (Tools|Quality Central). Follow the instructions to create your QC user account and report the bug. See the Borland Quality Central documentation for more information.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email [jpgpubs@borland.com](mailto:jpgpubs@borland.com). This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

# Creating JavaBeans with BeansExpress

This is a feature of  
JBuilder Developer  
and Enterprise

BeansExpress is the fastest way to create JavaBeans. It consists of a set of wizards, visual designers, and code samples that help you build JavaBeans rapidly and easily. Once you have a JavaBean, you can use BeansExpress to make changes to it. Or you can take an existing Java class and turn it into a JavaBean.

## What is a JavaBean?

---

A JavaBean is a collection of one or more Java classes, often bundled into a single JAR (Java Archive) file, that serves as a self-contained, reusable component. A JavaBean can be a discrete component used in building a user interface, or a non-UI component such as a data module or computation engine.

At its simplest, a JavaBean is a `public` Java class that has a constructor with no parameters. JavaBeans usually have properties, methods, and events that follow certain naming conventions (also known as design patterns).

## Why build JavaBeans?

---

Like other types of components, JavaBeans are reusable pieces of code that can be updated with minimal impact on the testing of the program they become a part of. JavaBeans have some unique advantages over other components, however:

- They are pure Java, cross-platform components.
- You can install them on the JBuilder component palette and use them in the construction of your program, or they can be used in other application builder tools for Java.
- They can be deployed in JAR files.

## Generating a bean class

---

To begin creating a JavaBean using BeansExpress,

- 1 Choose File|New Project and create a new project with the Project wizard.
- 2 Choose File|New (or click the New button on the JBuilder toolbar) to display the object gallery.
- 3 Select General to display the General page and double-click the JavaBean icon to open the JavaBean wizard.
- 4 Specify a name for the new JavaBean class in the Class Name field.
- 5 Specify the package you want the bean to be part of in the first text field. By default, this will be the name of your current project.
- 6 Choose a class to extend in the Base Class field.

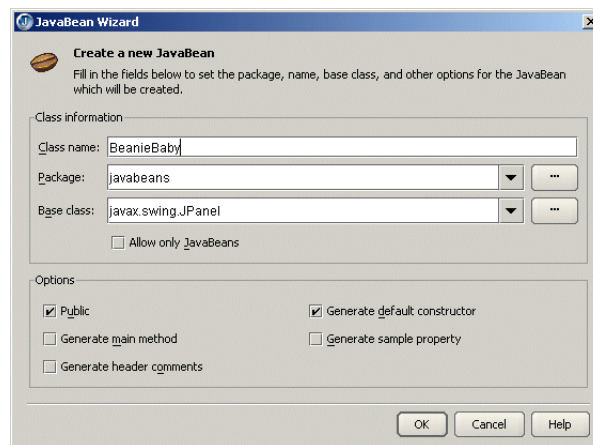
**Tip**

You can also start the JavaBean wizard by clicking the down arrow next to the New button on the toolbar, which displays a drop-down menu, and choosing JavaBean.

Either use the drop-down list, or click the adjacent button to display the Package browser and use it to specify any existing Java class you want.

- 7 Choose the remaining options you want; none of them are required:
  - a Check Allow Only JavaBeans if you want JBuilder to warn you if you try to extend a Java class that is not a valid JavaBean.
  - b Check Public if you want the class to be `public`.
  - c Check Generate Main Method if you want JBuilder to place the `main()` method within your bean and make it runnable.
  - d Check Generate Header Comments if you want Javadoc header comments (Title, Description, Author, and so on) added to the top of your class file.
  - e Check Generate Default Constructor if you want JBuilder to create a parameterless constructor.
  - f Check Generate Sample Property if you want JBuilder to add a property called `sample` to your bean. You might want this for your first bean to see how JBuilder generates the required code for a property. You can remove this property later, or make it an actual property that your bean can use.

Here is an image of the wizard with some typical values:





**8 Choose OK to close the JavaBean wizard.**

JBuilder creates a JavaBean with the name you specified, places it in your current project, and displays the source code it generated. This is the code JBuilder generates for the settings shown:

```
package javabeans;

import java.awt.BorderLayout;

import javax.swing.JPanel;

public class BeanieBaby extends JPanel {
    BorderLayout borderLayout1 = new BorderLayout();

    public BeanieBaby() {
        try {
            jbInit();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setLayout(borderLayout1);
    }
}
```

If you examine the code JBuilder generated, you'll see that

- JBuilder named the bean as you specified and extended the designated class; the class is declared as `public`.
- The class has a parameterless constructor.

Even in this rudimentary state, your class is a valid JavaBean.

## Designing the user interface of your bean

---

Not all JavaBeans have a user interface, but if yours does, you can use JBuilder's UI designer to create it.

To create a user interface for your bean,

- 1 Select the bean file JBuilder created for you in the project pane.
- 2 Click the Design tab to display the UI designer.
- 3 Use the UI designer to build the user interface for your bean.

For information on creating a user interface, see [Chapter 1, "Visual design in JBuilder."](#)

## Adding properties to your bean

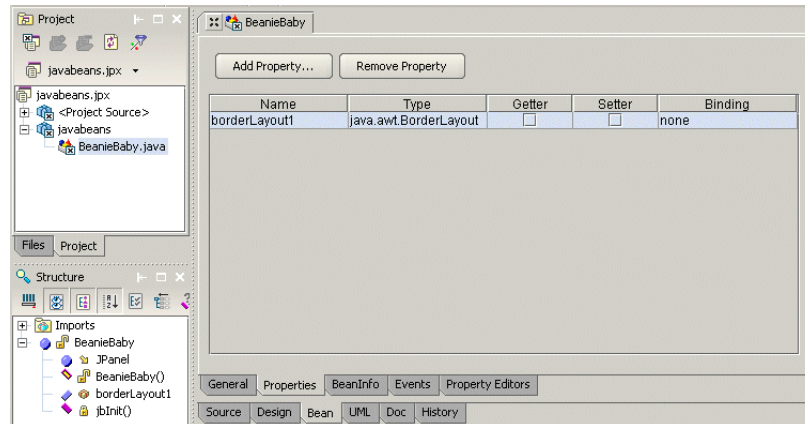
---

Properties define the attributes your bean has. For example, the `backgroundColor` property describes the color of the background.

JavaBeans properties usually have both a read and a write access method, also known as a getter and a setter, respectively. A getter method returns the current value of the property; a setter method sets the property to a new value.

To add a property to your bean,

- 1 Select your component in the project pane and click the Bean tab to display the BeansExpress designers.
- 2 Click the Properties tab to display the Properties designer.



- 3 Choose the Add Property button. The New Property dialog box appears.

- 4 Specify the name of the property in the Property Name box.

- 5 Specify a type in the Type box.

You can type in one of the Java types or use the Package browser to select any object type, including other JavaBeans.

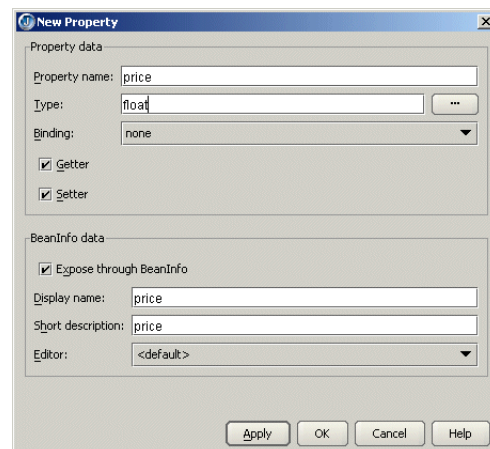
- 6 Leave both Getter and Setter check boxes checked if you want JBuilder to generate the methods to both get and set the property value.

If you want to make a property read-only, uncheck the Setter check box.

- 7 Choose OK.

JBuilder generates the necessary code for the property and adds the property to the Properties designer grid. You can see the read and write access methods added to the component tree for your bean. If you click the Source tab, you can see the code JBuilder generated.

Here is the New Property dialog box with all the required fields filled in:



The Display Name and Short Description are filled in automatically with default values. This is the resulting source code:

```
package javabeans;

import java.awt.BorderLayout;

import javax.swing.JPanel;

public class BeanieBaby extends JPanel {
    BorderLayout borderLayout1 = new BorderLayout();
    private float price;

    public BeanieBaby() {
        try {
            jbInit();
        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        setLayout(borderLayout1);
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public float getPrice() {
        return price;
    }
}
```

JBuilder added a `private price` field to the `BeanieBaby` class. The `price` field holds the value of the property. Usually a property field should be declared as `private` so that only the getter and setter methods can access the field.

It also generated a `setPrice()` method that can change the value of the new field, and it generated a `getPrice()` method that can get the current value of the field.

When your bean is installed on JBuilder's component palette and users drop the bean on the UI designer, the price property will appear in the Inspector so that users can modify its value. All the code to get and set the price property is in place.

## Modifying a property

---

Once you've added a property to your bean, you can modify it at any time with the Properties designer.

To modify a property,

- 1 Select your bean using the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Properties tab.
- 4 Select any of the fields in the Properties designer grid and make the changes you want.

For example, you can change the type of the property by entering another type.

JBuilder reflects the changes you make in the Properties designer in the source code of your bean. You can also make changes directly in the source code and the BeansExpress designers will reflect the changes if the changes have been made correctly.

## Removing a property

---

To remove a property from your bean,

- 1 Select the bean that contains the property in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Properties tab.
- 4 Select the property you want removed in the Properties designer grid.
- 5 Click Remove Property.

The property field and its access methods are removed from the source code.

## Adding bound and constrained properties

---

BeansExpress can generate the necessary code to create bound and constrained properties.

To add a bound or constrained property,

- 1 From the Properties designer, click the Add Property button to display the New Property dialog box.
- 2 Specify a property name and type.
- 3 Use the Binding drop-down list to specify the property as bound or constrained.
- 4 Choose OK.

If the property you added is a bound property, JBuilder adds the `private` property field to the class and generates its read and write methods. It also instantiates an instance of the `PropertyChangeSupport` class. For example, here is the code added for a bound property called `allTiedUp`:

```
public class BeanieBaby extends JPanel {
    ...
    private String allTiedUp;
    private transient PropertyChangeSupport propertyChangeListeners = new
        PropertyChangeSupport(this);
    ...
    public void setAllTiedUp(String allTiedUp) {
        String oldAllTiedUp = this.allTiedUp;
        this.allTiedUp = allTiedUp;
        propertyChangeListeners.firePropertyChange("allTiedUp",
            oldAllTiedUp, allTiedUp);
    }
    public String getAllTiedUp() {
        return allTiedUp;
    }
}
```

Note that the `setAllTiedUp()` method includes the code to notify all listening components of changes in the property value.

JBuilder also generates the event-listener registration methods that are called by listening components that want to be notified when the property value of `allTiedUp` changes:

```
public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
    super.removePropertyChangeListener(l);
    propertyChangeListeners.removePropertyChangeListener(l);
}
public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
    super.addPropertyChangeListener(l);
    propertyChangeListeners.addPropertyChangeListener(l);
}
```

The code JBuilder generates for a constrained property is similar except the listeners have the opportunity to reject the change in property value.

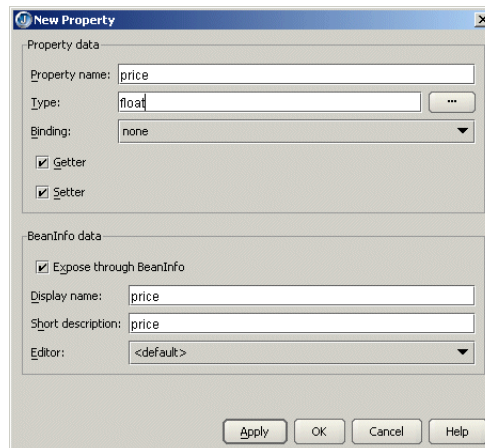
## Creating a BeanInfo class

You can customize how your bean appears in visual tools such as JBuilder using a `BeanInfo` class. For example, you might want to hide a few properties so they don't appear in JBuilder's Inspector. Such properties can still be accessed programmatically, but the user can't change their values at design time.

`BeanInfo` classes are optional. You can use BeansExpress to generate a `BeanInfo` class for you to customize your bean. If you are going to use a `BeanInfo` class, you should specify more detailed information for each new property you add to your bean.

## Specifying BeanInfo data for a property

You can specify `BeanInfo` data for a property in the New Property dialog box:



To hide a property so it does not appear in visual design tools such as JBuilder's Inspector, make sure the `Expose Through BeanInfo` option is unchecked.

To provide a localized display name for this property, enter the property name you want to use in the `Display Name` field.

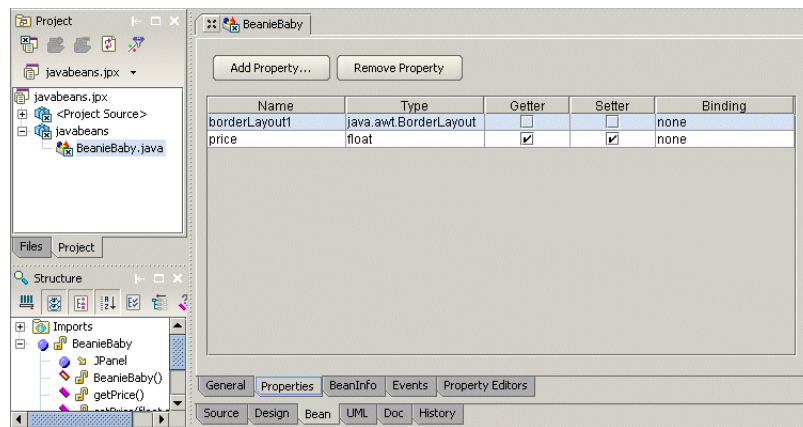
To provide a short description of this property, enter the description in the `Short Description` field. JBuilder displays this text as a tool tip in the Inspector for this property.

If you have created a property editor that can edit this field, specify the property editor in the `Editor` field. The `Editor` field displays all editors in scope that match the given property type.

## Working with the BeanInfo designer

The BeanInfo designer provides a way to modify BeanInfo data for a property, lets you specify the icon(s) you want to use to represent your bean in an application builder such as JBuilder, and generates the BeanInfo class for you.

Click the BeanInfo tab of the BeansExpress designer to open the BeanInfo designer.



The BeanInfo designer displays all the properties you have added to your bean. If you change your mind about the BeanInfo data you entered for a property, you can edit one or more of these fields in the grid. Only the name of the property can't be changed.

To provide an image to use as an icon to represent your bean, specify one or more images in the Icons panel.

You can specify different icons to use for both color and mono environments and for different screen resolutions. Fill in the boxes that meet your needs.

If the superclass of your bean has a BeanInfo class and you want its BeanInfo data exposed in the BeanInfo class for your bean, check the Expose Superclass BeanInfo check box. The generated code will include a `getAdditionalBeanInfo()` method that returns the BeanInfo data of the class your bean extends.

To generate a BeanInfo class for your bean, click the Generate Bean Info button.

JBuilder creates a BeanInfo class for you. You'll see it appear in the project pane. To see the generated code, select the new BeanInfo class in the project pane and the source code appears.

## Modifying a BeanInfo class

You can change the BeanInfo class for your bean with BeansExpress.

- 1 Select your bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the BeanInfo tab to display the BeanInfo designer.
- 4 Make your changes in the grid.
- 5 Click the Generate Bean Info button again.

You are warned that you are about to overwrite the existing BeanInfo class. If you choose OK, the class is overwritten with the new BeanInfo data.

## Adding events to your bean

A JavaBean can

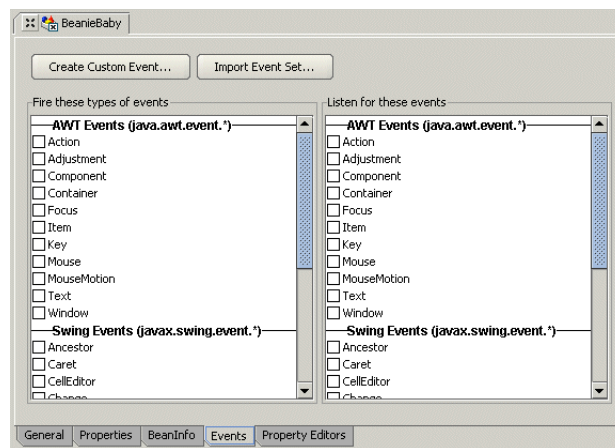
- Generate (or fire) events, sending an event object to a listening object.
- Listen for events and respond to them when they occur.

BeansExpress can generate the code that makes your bean capable of doing one or both of these things.

### Firing events

To make your bean capable of sending an event object to listening components,

- 1 Select your bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Events tab to display the Events designer.



- 4 Select the events you want your bean capable of firing in the left window.

The Events designer adds the event-registration methods to your bean. These methods are called by components that want to be notified when these types of events occur. For example, if you select Key events, these methods are added to your bean:

```
package myjavabean;

import java.io.*;
import java.beans.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.util.*;

public class BeanieBaby extends JPanel {
    BorderLayout borderLayout1 = new BorderLayout();
    private float price;
    private transient Vector keyListeners;
```

```

public BeanieBaby() {
    try {
        jbInit();
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}

private void jbInit() throws Exception {
    this.setLayout(borderLayout1);
}

public void setPrice(float price) {
    this.price = price;
}

public float getPrice() {
    return price;
}

transient Vector keyListeners;

public synchronized void addKeyListener(KeyListener l) {
}

public synchronized void removeKeyListener(KeyListener l) {
}

...
}

```

When a component wants to be notified of a key event occurring in your bean, it calls the `addKeyListener()` method of your bean and that component is added as an element in `keyListeners`. When a key event occurs in your bean, all listening components stored in `keyListeners` are notified.

The class also generates `fire<event>` methods that send an event to all registered listeners. One such event is generated for each method in the `KeyListener` interface. For example, the `KeyListener` interface has three methods: `keyTyped()`, `keyPressed()`, and `keyReleased()`. So the Events designer adds these three `fire<event>` methods to your bean class:

```

protected void fireKeyTyped(KeyEvent e) {
    if(keyListeners != null) {
        Vector listeners = keyListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((KeyListener) listeners.elementAt(i)).keyTyped(e);
        }
    }
}

protected void fireKeyPressed(KeyEvent e) {
    if(keyListeners != null) {
        Vector listeners = keyListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((KeyListener) listeners.elementAt(i)).keyPressed(e);
        }
    }
}

```



```

    }
}

protected void fireKeyReleased(KeyEvent e) {
    if(keyListeners != null) {
        Vector listeners = keyListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((KeyListener) listeners.elementAt(i)).keyReleased(e);
        }
    }
}
}
}

```

Once you've made your bean capable of generating events, those events will appear in JBuilder's Inspector when the user drops your bean on the UI designer.

## Listening for events

---

You can also make your bean a listener for events that occur in other components.

To make your bean a listener, select one of the event types to listen for in the Events designer.

As soon as you check one of the event types, the Events designer implements the associated listener interface in your bean. For example, if you checked `java.awt.event.KeyListener`, the phrase `implements KeyListener` is added to the declaration of your bean and the `KeyPressed()`, `KeyReleased()`, and `KeyTyped()` methods are implemented with empty bodies.

Here is a bean that includes the generated code to implement the `KeyListener` interface:

```

package myBeans;

import java.awt.*;
import javax.swing.JPanel;
import java.beans.*;
import java.awt.event.*;

public class JellyBean extends JPanel implements KeyListener { // implements KeyListener
    BorderLayout borderLayout1 = new BorderLayout();

    public JellyBean() {

        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    void jbInit() throws Exception {
        this.setLayout (borderLayout1);
    }

    public void keyTyped(KeyEvent e) {           // Adds new method
    }

    public void keyPressed(KeyEvent e) {         // Adds new method
    }

    public void keyReleased(KeyEvent e) {        // Adds new method
    }
}

```

If your bean is registered with another component as a listener (by calling the event-registration method of the component), the source component calls one of the methods in the listening component when that type of event occurs. For example, if a `KeyPressed` event occurs in the source component, the `KeyPressed()` method is called in the listening component. Therefore, if you want your component to respond in some way to such an event, write the code that responds within the body of the `KeyPressed()` method, for example.

## Creating a custom event set

Occasionally you might want to create a custom event set to describe other events that can occur in your bean. For example, if your bean implements a password dialog box, you might want the bean to fire an event when a user successfully enters the password. You also might want the bean to fire another event when the user enters the wrong password. Using BeansExpress, you can create the custom event set that handles these situations.

To create a custom event set,

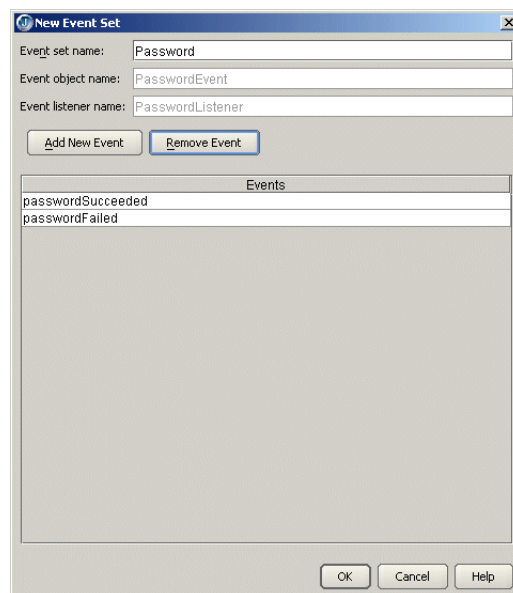
- 1 Select a bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Events tab.
- 4 Click the Create Custom Event button.

The New Event Set dialog box appears.

- 5 Specify the name of the event set in the Name Of New Event Set box.

The names of the event object and the event listener are generated from the name of the event set; they are displayed in the dialog box.

- 6 Select the `dataChanged` item and change it to the name of your first event.
- 7 To add more events, choose the Add New Event button for each event and change the added item to the names of your remaining events:



**8 Choose OK.**

The new event set is added to the list of event types that can be fired in the Events designer.

JBuilder generates the event object class for the event set:

```
package javabeans;

import java.util.*;

public class PasswordEvent extends EventObject {
    public PasswordEvent(Object source) {
        super(source);
    }
}
```

JBuilder also generates the Listener interface for the event set:

```
package javabeans;

import java.util.*;

public interface PasswordListener extends EventListener {
    public void passwordSuccessful(PasswordEvent e);
    public void passwordFailed(PasswordEvent e);
}
```

## Creating a property editor

---

A property editor is an editor for changing property values at design time. You can see several different types of property editors in JBuilder's Inspector. For example, for some properties, you simply type in a value in the Inspector, and by so doing, change the value of the property. This is the simplest type of property editor. For other properties, you use a choice menu (drop-down list) to display all the possible values and you select the value you want from that list. Colors and fonts have property editors that are actually dialog boxes you can use to set their values.

When you create your own JavaBeans, you might create new property classes and want to have editors capable of editing their values. BeansExpress can help you create property editors that display a list of choices.

To begin creating a property editor for a property,

- 1 Click the Property Editors tab in BeansExpress.
- 2 Click the Create Custom Editor button.

The New Property Editor dialog box appears.

- 3 Specify the name of your property editor in the Editor Name box. Give the property editor the same name as the property it will edit with the word Editor appended. For example, if the property name is `favoriteFoods`, name the editor `FavoriteFoodsEditor`.

- 4 Select the type of editor you want from the Editor Type drop-down list.

The appearance of the New Property dialog box changes depending on which type of editor you selected. The next four sections describe how to create a property editor of the four different types.

## Creating a String List editor

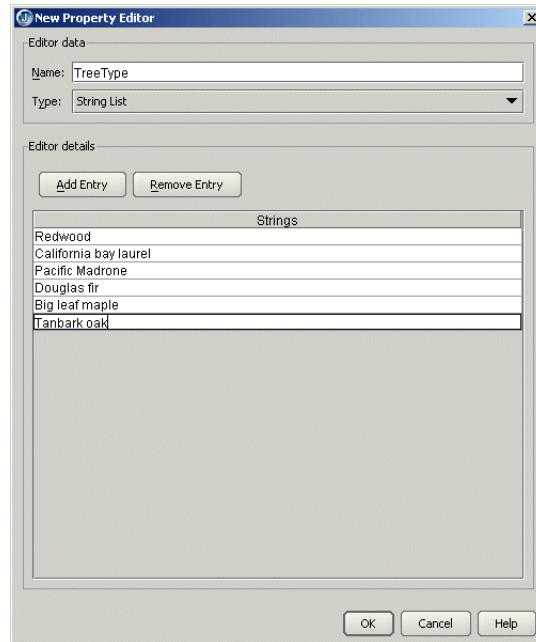
---

A String List is a simple list of Strings. It appears in the Inspector as a drop-down list containing the strings you specify. When the user uses the list to select an entry, the property being edited is set to the selected value.

To add items to a String List editor,

- 1 Choose Add Entry for each item you want to appear in the list.
- 2 In each entry, enter the string you want to appear.

This is how the resulting New Property Editor dialog box might look:



- 3 Choose OK, and a new property editor class is created.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Creating a String Tag List editor

---

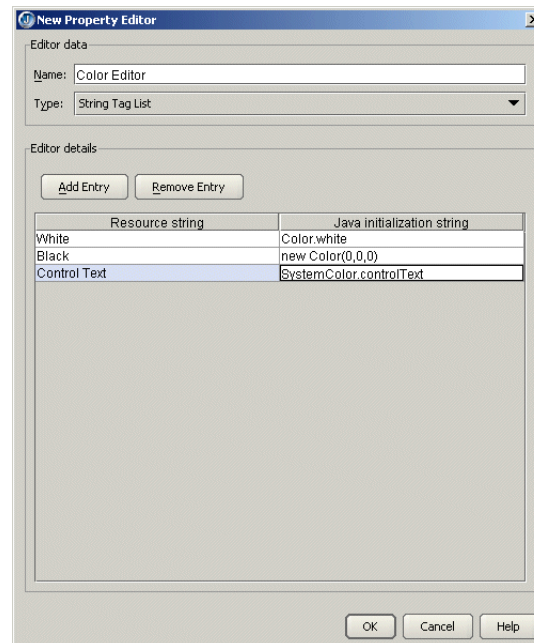
A property editor that is a String Tag List also presents a list of strings to the user in the Inspector. When the user selects one of the items, the specified Java initialization string is used to set the property. A Java initialization string is the string JBuilder uses in the source code it generates for the property.

To add items to a String Tag List editor,

- 1 Choose Add Entry for each item you want to appear in the list.
- 2 In each entry, enter the string you want to appear and its associated Java initialization string.

If you want to include a string in your list of Java initialization strings, put quotation marks (") before and after the string, as if you were entering it in source code.

Here is an example of how the dialog box might look:



- 3 Choose OK, and a new property editor class is created.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Creating an Integer Tag List editor

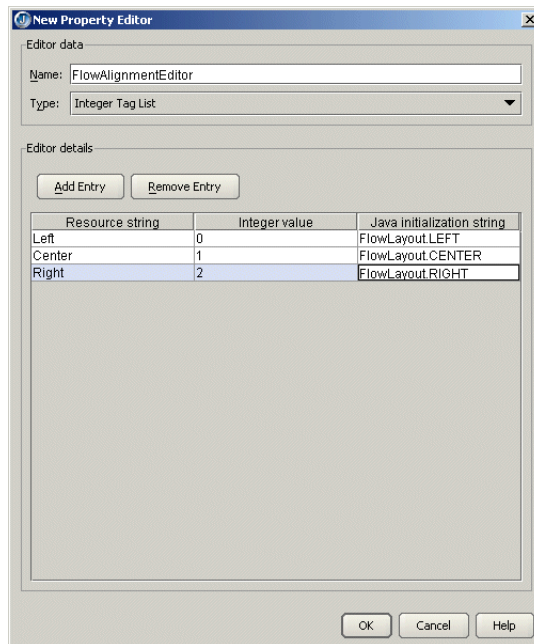
An Integer Tag List property editor can be used to edit integer properties. It presents a list of strings to the user in the Inspector, and when the user selects one of the items, the specified Java initialization string is used to set the property.

To add items to an Integer Tag List editor,

- 1 Choose Add Entry for each item you want to appear in the list.
- 2 In each entry, enter the string you want to appear and its associated integer value and Java initialization string.

If you want to include a string in your list of Java initialization strings, put quotation marks (") before and after the string, as if you were entering it in source code.

Here is an example of how the dialog box might look:



- 3 Choose OK, and a new property editor class is created.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Creating a custom component property editor

You can also use your own custom component to edit the value of a property. Selecting this choice generates a skeleton property editor that uses the custom editor you specify to actually edit the property.

To specify a custom component property editor,

- 1 Enter a name for the editor class that will instantiate your custom component in the Editor Name field of the New Property dialog box.
- 2 Select Custom Editor Component from the drop-down list.
- 3 Specify the name of your custom component as the value of the Custom Component Editor field.
- 4 Check the Support paintValue() option if your custom editor paints itself.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Adding support for serialization

Serializing a bean saves its state as a sequence of bytes that can be sent over a network or saved to a file. BeansExpress can add the support for serialization to your class.

To add support for serialization,

- 1 Select your bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the General tab to display the General page.
- 4 Check the Support Serialization option.

BeansExpress modifies the class so that it implements the `Serializable` interface. Two methods are added to the class: `readObject()` and `writeObject()`:

```
private void readObject(ObjectInputStream ois) throws IOException,
    ClassNotFoundException {
    ois.defaultReadObject();
}

private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}
```

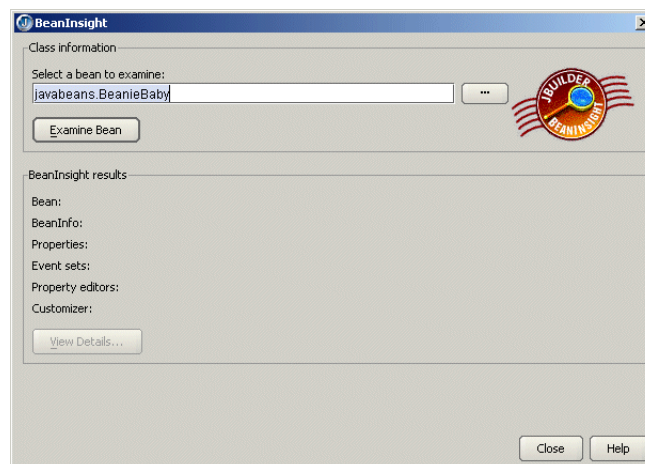
You must fill in these two methods to serialize and deserialize your bean.

## Checking the validity of a JavaBean

When you're finished with your bean, you can use BeanInsight to verify that the component you created is a valid JavaBean component. If your class fails as a JavaBean, BeanInsight reports why it failed. It identifies all the properties, customizers, and property editors it finds for the class. It also reports whether the property information is obtained through a BeanInfo class or through introspection.

To verify that your Java class is a JavaBean,

- 1 Choose Tools|BeanInsight to display BeanInsight:



- 2 Type in the name of the component you want examined or use the ellipsis (...) button to specify the component. If the bean is already selected in the project pane when BeanInsight appears, its name is displayed in BeanInsight.

- 3 Click the Examine Bean button to begin the examination process.  
BeanInsight reports a summary of its findings in the BeanInsight Results section of the wizard.
- 4 To see complete details on BeanInsight's findings, click the View Details button.
- 5 Click the various tabs to see the full report.

## Installing a bean on the component palette

---

Once you have built a valid JavaBean component, you are ready to install it on the component palette using the Palette Properties dialog box. The class files for your new component must be on your classpath.

To display the Palette Properties dialog box, choose Tools|Configure|Palette or right-click the component palette and choose Properties.

For information about installing components, click the Help button of the Palette Properties dialog box or see [“Adding a component to the component palette” on page 74](#).



## Introducing the visual designer

The visual designer consists of features that allow you to visually design classes containing default public constructors. The “visual designer” is a collective term for several design tools that are tailored to do different types of design. These include the UI designer, the Menu designer, the Data Access designer, and the Default designer. The Default designer is for components that don’t fit into any of the other three categories. When you access the visual designer, JBuilder opens the type of designer that’s appropriate for the active file.

The visual designer is an OpenTool. Advanced users who want to learn to add a designer type or otherwise customize the designer can use the OpenTools documentation to learn how to do so.

### Using the designer

---

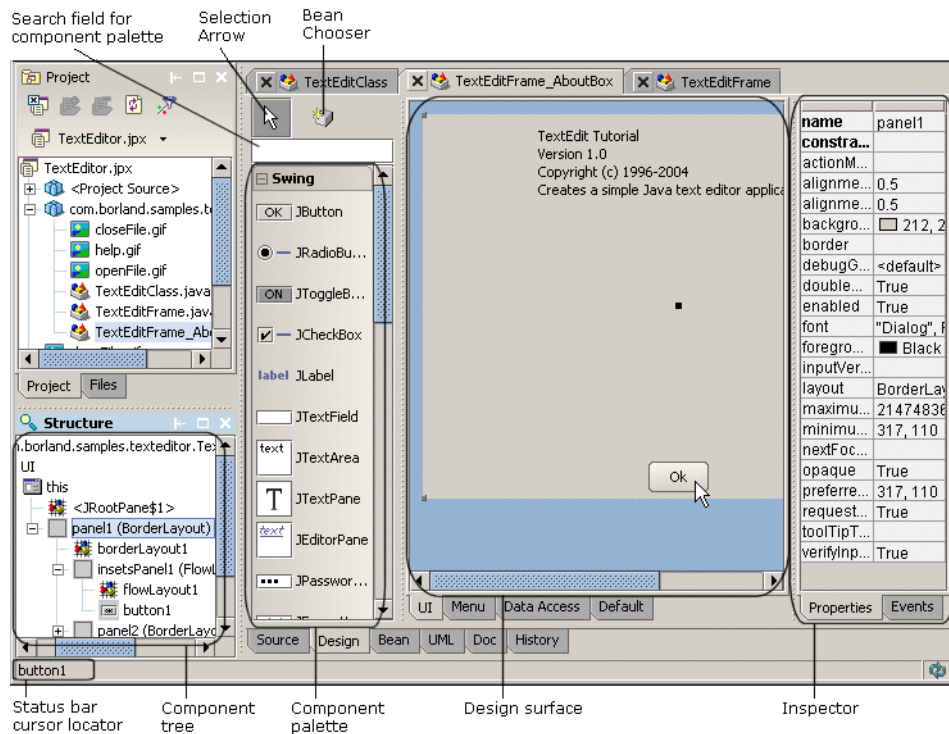
Click the Design tab of the content pane to access the visual designer. When the designer is active, the working areas of the IDE change to accommodate design tasks:

- The content pane shows the design surface.

The content pane also contains:

- The Search field for the component palette, the Selection Arrow, and the Bean Chooser, displayed at the top edge of the component palette.
- The component palette, displayed at the left of the content pane.
- The Inspector, displayed on the right margin of the content pane.
- The available designer types in tabs along the bottom of the content pane.
- The structure pane shows the component tree.

The component that’s selected, either in the component tree or on the design surface, is highlighted in the component tree and reflected in the Inspector. The status bar indicates the component the cursor rests on, on the design surface. This allows you to work with one component while you locate others, without changing focus until you want to.

**Figure 3.1** The designer

In this example,

- The `panel1` component is selected, making it the active component.

We can tell because it's highlighted in the component tree, its handles or nibs are visible on the design surface, and it's described in the Inspector.

- The pointer is on the design surface, hovering over the `button1` component.

We can tell because the name of the component appears in the status bar.

If this user clicked the mouse in its present location, then `button1` would be selected, highlighted in the component tree, and reflected in the Inspector, as well as displayed in the status bar.

JBuilder's Two-Way Tools technology keeps the designer and the source code synchronized. It immediately changes the code according to changes made in the designer, and changes the design in the designer to reflect changes made in the code.

## The design surface

The design surface is your virtual sketch pad. You can add or remove components directly on it, edit the size of components, and see what your overall design looks like as it evolves.

Select a component on the design surface or in the component tree to make it appear in the Inspector, where you can edit its properties.

Select a container to nest another component inside it or to activate its handles. In most layout managers, you can grab handles with your mouse to change a component's size or location.

**Tip** When you move a component on the design surface, a tool tip appears displaying the name of the component and real-time position information. The information provided depends on the layout manager used by the component.

All properties that can be changed on the design surface can also be changed in the Inspector.

### See also

- [Chapter 9, “Using layout managers”](#)
- [Chapter 4, “Using the component tree and Inspector”](#)
- [Chapter 5, “Handling events”](#)

### Pinpointing a component

Within a complex design, it can be difficult to tell exactly which of several likely components your cursor is on, on the design surface. The status bar eliminates all confusion.

As you move the mouse pointer over a component on the design surface, the status bar at the bottom of the IDE displays the name of the component. If the panel containing the component is in `XYLayout`, the status bar also displays the x,y coordinates.



A screenshot of the status bar in an IDE. It shows a tooltip for a component named 'JPanel1' which is using 'XYLayout'. The tooltip also displays the component's coordinates: 'x: 83 y: 128'.

When in doubt, use the component tree to select components in the design.

### See also

- [“Selecting components in the UI” on page 54](#) for more on handling components on the design surface

## The component palette

---

The component palette provides quick access to all available component libraries. Rest your pointer over a component’s icon to display a tool tip with the component’s name. The component palette is customizable: you can add new components and new pages to it.

Rest your pointer on a page in the component palette to see the page’s tool tip. The tool tip displays the page name, the number of components on it, and the description of that page.

Add a component to your design in any of these ways:

- Choose the page containing the types of components you want. Select a component in the component palette and click on the design surface where you want the component’s upper left corner to be.

The component drops onto the design surface. Its upper left corner lands where you clicked. The component also appears in the component tree.

- Choose the page containing the types of components you want. Select a component in the component palette and click in the appropriate hierarchical location in the component tree.

The component drops into the component tree. If it’s a visible component, it’s visible in the appropriate place on the design surface.

- Choose Edit|Add Component.

The Add Component dialog box appears. Select a component library and choose a component, then click OK.

The component appears in the component tree. If it’s a visible component, it appears in the appropriate place on the design surface.

### See also

- [“Adding components” on page 36](#)
- [“Managing the component palette” on page 73](#)

## Using the Selection Arrow



The Selection Arrow button lets you know when you have a selection loaded vs. when you are free to make a new selection. When the Selection Arrow button appears depressed, you are free to make a selection. When you have selected something in the component palette, and can still drop it into the component tree or on the design surface, the Selection Arrow button is not depressed.

When you choose a component from the component palette or the Add Component dialog box, the Selection Arrow button automatically depresses, letting you know that your selected component is loaded. When you have a selection loaded but you don't want to use it or have finished using it, click the Selection Arrow to deselect the loaded component.

By default, the Selection Arrow only holds the selection until you drop it once. It's automatically reset after you drop the selected component.

To use multiple instances of the same component, hold down the *Shift* key when you click the component in the component palette. You can drop the component onto the design surface as often as you want (creating a new instance each time) and the Selection Arrow stays flush, letting you know that the cursor is still loaded with that component. When you have enough instances of the component, click the Selection Arrow to deselect the loaded component.

## Using the Bean Chooser



The Bean Chooser button is at the top edge of the component palette beside the Selection Arrow. It displays a user-defined list of beans. When you choose a bean from the list, the Bean Chooser loads your mouse cursor with a reference to the bean, just as if you had clicked a component on the palette. When you click the design surface, it adds the bean you chose.

To add a bean to the Bean Chooser drop-down list,

- 1 Make sure the library that contains the bean is listed as a required library for your project in the Project Properties dialog box.

If not, then add the library.

- 2 Click the Bean Chooser button and choose Select from the menu.

This displays the Bean Chooser dialog box.

- 3 Use either the Search page or the Browse page:

- In the Search page, start typing the bean name in the Search For field.
- In the Browse page, either start typing the fully qualified name or expand the nodes until you locate the bean you want.

The Search page can find the bean using only its short name. The Browse page requires you to select from the tree or to type in the fully-qualified class name.

- 4 Select the bean and click OK.
- 5 Click the Bean Chooser button again.

A drop-down menu now appears containing the new JavaBean.

When you want to use that bean when working on the same project, click the Bean Chooser button and select the bean from the menu.

**See also**

- “Adding and configuring libraries” in *Building Applications with JBuilder*

## The Inspector

---

The Inspector displays the properties and events of the selected component, and provides right-click context menus, editable text fields, and other controls to allow you to edit properties and events. Custom editors can be used along with the Inspector. Select a component in the component tree or on the design surface to display its properties and events in the Inspector.

Click the Properties tab to view and edit a component’s properties. Click the Events tab to view and edit a component’s events.

Adjust the size of the Inspector by dragging the left border of the Inspector.

**See also**

- [“Using the Inspector” on page 39](#)
- [“Using customizers in the designer” on page 79](#)

## The component tree

---

The component tree provides access to each type of available designer and provides a hierarchical view of the components in the active file. It also acts as a component manager, allowing you to add and remove components and rearrange the components in the design hierarchy.

When you click the Design tab, the designer automatically opens to the type of designer appropriate to the active file’s outer container. You may want to choose other designers to work on components either within, or referenced by, the active file. Select a designer type by selecting the appropriate tab: UI, Menu, Data Access, or Default. Alternatively, you can switch designer types by double-clicking a component within that type’s folder in the component tree.

Select a component in the component tree to put focus on it on the design surface and to display that component’s properties and events in the Inspector. Move components within the component tree using either the mouse or the keyboard.

**See also**

- [“Using the component tree” on page 35](#)

## Designer categories

---

JBuilder provides individual designers for four broad categories of JavaBeans:

- UI designer
- Menu designer
- Data Access designer
- Default designer

Each of these designers provides a set of features that makes designing its particular type of component easier. Switch between designer types in one of these ways:

- Click the tab of the designer type at the bottom of the content pane.
- Double-click a component of the designer type in the component tree.
- Select a component of the designer type in the component tree and press *Enter*.

## UI designer

---

UI components are the elements of the program that the user can see and interact with at runtime. They derive ultimately from `java.awt.Component`. In the designer, UI components that are normally visible at runtime appear in the UI designer. UI components that normally don't show, such as pop-up menus, appear in the Default folder of the component tree.

Whenever possible, JBuilder's UI components are "live" at design time. For example, a list displays its list of items, or a data grid connected to an active data set displays current data.

### See also

- [Chapter 6, "Creating user interfaces"](#)
- "Tutorial: Creating a simple user interface" in online help

## Menu designer

---

Menu components derive from `java.awt.MenuComponent`. At design time, JBuilder displays menu components in the Menu folder in the component tree.

Menu work appears on the design surface only when you're using the Menu designer. For instance, it doesn't show when you use the UI designer. Your menu work is still exactly the way you left it; it just doesn't get displayed in the other designers.

### See also

- [Chapter 7, "Designing menus"](#)

## Data Access designer

---

Data access components are non-UI components used in a database application to connect controls to data. Data access components do not appear in the UI container at design time, but they do appear in the component tree in the `Data Access` folder. Select a data access component in the component tree to make its properties and events accessible in the Inspector.

### See also

- ["Database components" on page 59](#)
- "DataExpress components" in *Developing Database Applications*

## Default designer

---

The default designer provides visual design tools for components that aren't surface UI, menu, or data access components. Examples of these may include pop-up UI elements, such as dialog boxes, or non-UI JavaBean components such as `buttonGroup`. Click the Default tab or select these components from the Default folder in the component tree to activate the default designer.

Once you understand how to use the component tree, the design surface, the component palette, and the Inspector, you know how to use the default designer.

## Keyboarding in the designer

---

There are two types of keyboard shortcuts: navigational shortcuts that move focus from one area to another, and action shortcuts, usually involving the keyboard and mouse, that simplify working in the designer.

### Navigational shortcuts

Use either the mouse, the *Tab* key alone, the *Ctrl+Tab* keys, or the arrow keys to navigate through the designer. Combine these keystrokes with the *Shift* key to move focus in reverse order.

### Action shortcuts

Below are keyboard/mouse shortcuts that facilitate working in the designer:

| Keystroke                     | Action                                                                                                                                      |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Ctrl+Click</i>             | Individually select/deselect multiple components in component tree or on the design surface.                                                |
| <i>Shift+Click</i>            | Drop multiple instances of a component onto the design surface. Use the Selection Arrow on the component palette to turn off the selection. |
| <i>Shift+F10</i>              | Display the design surface's useful context menu for the component selected in the component tree.                                          |
| <i>Ctrl+X</i>                 | Cut a selection from the design surface or the component tree to the Clipboard.                                                             |
| <i>Ctrl+C</i>                 | Copy a selection to the Clipboard from the design surface or the component tree.                                                            |
| <i>Ctrl+V</i>                 | Paste the contents of the Clipboard onto the design surface or the component tree at the location of the cursor.                            |
| <i>Ctrl+Arrow</i>             | Move the selected component one pixel in the direction of the arrow.                                                                        |
| <i>Ctrl+Shift+Arrow</i>       | Move the selected component eight pixels in the direction of the arrow.                                                                     |
| <i>Ctrl+Z</i>                 | Undo the most recent action. Can use repeatedly for undoing multiple successive actions.                                                    |
| <i>Ctrl+Shift+Z</i>           | Redo the most recent undo. Can use repeatedly for redoing multiple successive undo's.                                                       |
| <i>Ctrl+Del</i> or <i>Del</i> | Delete the selection.                                                                                                                       |
| <i>Shift+Drag</i>             | Drag a rectangle around multiple components to select them on the design surface.                                                           |
| <i>Shift+Drag</i>             | Limit the move to orthogonal directions (up, down, right, left or at 45 degree angles).                                                     |
| <i>Alt+Drag</i>               | Drag a component into a parent container.                                                                                                   |
| <i>Click+Drag</i>             | Drag the component to a new location.                                                                                                       |
| <i>Ctrl+Drag</i>              | Drag a copy of the selected object to a new location.                                                                                       |





## Using the component tree and Inspector

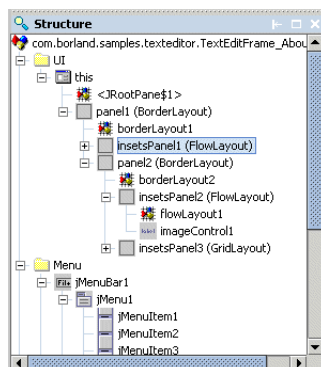
The component tree and the Inspector provide access to all the components and all component properties and events in the active file. The component tree shows which designer you're using, what components are in the active file, and which component is selected. The Inspector displays the properties and events belonging to the selected component. Select a component either on the design surface or in the component tree to view its properties in the Inspector. Rename a component in either the component tree or the Inspector.

### Using the component tree

---

The component tree allows you to view and manage the components in a visually designable file. It shows all of the components in the active file and their relationships, the layout managers associated with UI containers, and the type of designer each component uses. It provides access to commands and controls for the designers and the components. Changes made in the component tree are immediately reflected in the Inspector, the design surface, and the source code.

The component tree always shows exactly which component is selected, making accurate selection easy regardless of the type of component or the complexity of design.



The component tree allows you to view and manage designer types and components:

- Open a particular designer, such as the Menu designer.
- Add components to the class from the component palette.
- See a component's name.
- Select a component in the component tree to modify its properties and events in the Inspector.
- Select a component in the component tree to modify it on the design surface.
- Change the name of a component.
- Move a component to a different container or a different place in the hierarchy.



Before selecting existing components, be sure the Selection Arrow button on the component palette appears depressed. Otherwise you may accidentally place a new component on your design.

The component tree supports multiple selection:

- Use the *Ctrl* key and the cursor to add individual selections.
- Use the *Shift* key and the cursor to add contiguous selections.
- Hold down the left mouse button and draw a rectangle around the group of components you want to change.

Control the cursor either with the mouse or with the arrow keys.

## Opening particular designer types

---

Either click the appropriate tab in the content pane, or double-click the appropriate folder or any node inside it in the component tree.

For instance, with any other designer active, expand the Menu folder and select a `menuBar` component inside it to access the Menu designer. Menu components become available and menu-specific commands in the designer appear.

## Adding components

---

Components can be added in either of two ways: using the mouse to drag and drop, or using the menus and keyboard to select.

### Using menu commands

To add a component using menu commands

- 1 Select a parent node in the component tree.
- 2 Choose Edit|Add Component.

The Add Component dialog appears.

A list of component libraries is on the left. A list of the components available in the selected library appear on the right.

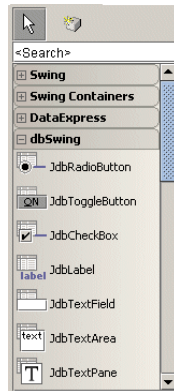
- 3 Select or type in the component library and the component you want.
- 4 Click OK.

The component appears under the node you originally selected in the component tree.

## Using the mouse

To add a component using the mouse

- 1 Click a component on the component palette.



- 2 Do one of the following. Choose a technique based on the type of component selected:

- *All components.* Click the target container in the component tree.
- *Visible components.* Click on the design surface.

In `null` or `XYLayout`, the upper-left corner of the component is anchored where you clicked.

An adjustable component appears at its default size.

- *Visible components.* Click on the design surface and, holding the mouse button down, drag down or right to the desired size.

In `null` or `XYLayout`, the upper-left corner of the component is anchored where you clicked.

**Note** Ultimately, the layout manager for each container in your UI will determine its components' sizes and positions.

To add multiple instances of a component,

- 1 Press the *Shift* key while clicking a component on the component palette.
- 2 Click repeatedly on the design surface or in the component tree to add multiple instances of the component.



- 3 Clear the selection when you are done by clicking the component palette's Selection Arrow button or by choosing another component on the palette.

**Note** Be sure to clear the selection. Otherwise, you may inadvertently create an extra, unwanted component.

## See also

- [Chapter 9, "Using layout managers"](#) for more information on `null`, `XYLayout`, and other layout managers

## Cutting, copying, and pasting components

To cut, copy, or paste selected components in the designer,

- Right-click the selected components, then choose Cut, Copy, or Paste from the context menu.
- Choose the command you want (Cut, Copy, or Paste) from the Edit menu.

- Use the appropriate shortcut keys for the function. In most keymappings, these are
  - Cut: *Ctrl+x*
  - Copy: *Ctrl+c*
  - Paste: *Ctrl+v*

## Deleting components

---

To delete a component, select the component on the design surface or the component tree. Then, either choose *Edit|Delete*, use the keyboard shortcut defined for your keymap, or press the *Del* key.

## Using Undo and Redo

---

To undo or redo an action in the designer, do one of the following:

- Right-click anywhere on the design surface or the component tree, and choose Undo or Redo from the context menu.
- Click anywhere on the design surface or the component tree and choose *Edit|Undo* (usually *Ctrl+Z*) or *Edit|Redo* (usually *Ctrl+Shift+Z*).

You can undo multiple successive actions by choosing Undo repeatedly. This undoes your changes by stepping back through your actions and reverting your design through its previous states.

Redo reverses the effects of your last Undo. You can redo multiple successive actions by choosing Redo repeatedly. Redo is available only after an Undo command.

## Changing a component name

---

You can change the name of a component by using the component tree. When you do so, JBuilder performs a rename refactor on the component name.

To change the name in the component tree:

- 1 Select the component in the component tree.
- 2 Make the component name editable in one of the following ways:
  - Press *F2*.
  - Right-click the component name in the tree and choose *Rename*.
- 3 Type the text for the new name.
- 4 Press *Enter*.

The new name appears in place of the old one, and all references to that component are correctly updated.

## Moving components

---

Using the mouse, drag and drop a component in the component tree to move it.

Using the keyboard

- 1 Select the component to be moved.
- 2 Cut it.
  - Either right-click it and choose *Cut*, or choose *Edit|Cut*.





- 3 Select the component or folder immediately above where you want to paste the component.
- 4 Paste the cut component in.  
Either right-click and choose Paste, or choose Edit|Paste.

## Viewing component class names

Put your cursor over the component name. The class name appears in a tool tip.

## Understanding component tree icons

Icons for individual components (panes, buttons, and so on) are generally smaller versions of the icons used in the component palette. More general icons appearing in the component tree include:

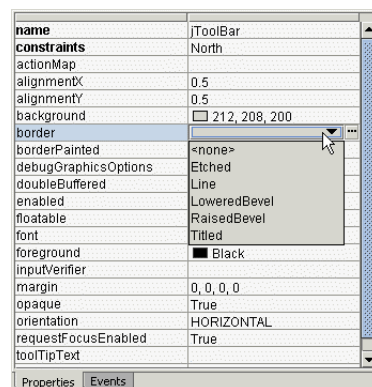
| Icon                                                                              | Explanation                                                             |
|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------|
|  | The current file.                                                       |
|  | The layout manager for the parent container.                            |
|  | The default icon used for a component that doesn't define its own icon. |
|  | Designer type. Can be UI, Menu, Data Access, Default, or custom.        |

## Using the Inspector

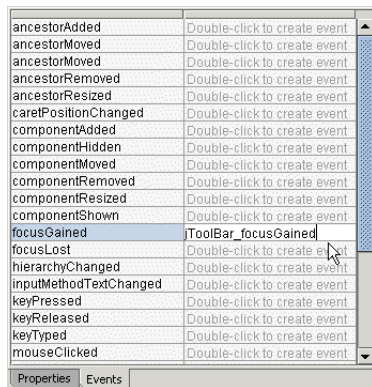
The Inspector appears in the right side of the content pane in the designer. It lets you visually edit component properties and attach handlers to component events. Select a component in either the design surface or the component tree to display its attributes in the Inspector. The Inspector's left column shows the names of the component properties or events. The right column shows their current values.

The Inspector has two tabs: Properties and Events.

View and edit properties on the Properties tab:



View and edit events on the Events tab:



Using the Inspector, you can

- Change properties' exposure levels by right-clicking the property and choosing Property Exposure Level.
- Change property values by clicking in the right-hand column and either using the drop-down lists or typing into the space provided.
- Set the initial property values for components in the container, and for the container and its layout manager, by choosing or entering values in the right-hand Properties column.
- Create event handling code by double-clicking the right-hand column in the Events tab. This creates code to catch events in a container receiving events from a component inside the container.
- Localize strings by right-clicking a property that takes a `String` value and choosing ResourceBundle.

Any changes you make in the Inspector are reflected immediately in the source code and in the rest of the designer.

### See also

- [Chapter 5, "Handling events"](#)
- ["Using customizers in the designer" on page 79](#) to learn how customizers work with the Inspector
- ["Tutorial: Building a Java text editor"](#) in online help for practice using the Inspector

## Surfacing property values

In order to be surfaced in the Inspector, a property value must

- Be a class-level variable.
- Have a Hidden or Expert exposure level.

### Making properties class variables

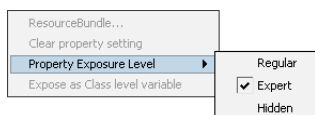
Properties that are `static` variables are editable in the Inspector. To make an instance variable a class-level variable using the Inspector,

- 1 Right-click the property in the Inspector.
- 2 Choose Expose As Class Level Variable from the context menu.

The pertinent value appears in the Inspector. JBuilder writes a new variable declaration and applies the value to it. This way you can manipulate the value outside the context of the property.

## Setting property exposure

You can choose what level of properties are exposed for the component in the Inspector based on how the properties are marked in the component's BeanInfo class. Right-click in the Inspector and choose Property Exposure Level to display a context menu with three choices:



- **Regular:** The Inspector displays only the properties marked Regular, not those marked Hidden or Expert.
- **Expert:** The Inspector displays the properties marked Regular and Expert.
- **Hidden:** The Inspector displays all properties: Regular, Expert, and Hidden.

**Note** In order to be seen, properties must either be exposed in the BeanInfo class or else set in the `jbInit()` method for the class.

## Setting property values

Properties are attributes that define how a component appears and responds at runtime. In JBuilder, you set a component's initial properties during design time, and your code can change those properties at runtime.

The Properties page in the Inspector displays the properties of the selected components. This is where you set the property values at design time for any component in your design. By setting properties at design time, you are defining the initial state of a component when the UI is instantiated at runtime.

**Note** To modify property values at runtime, you can put code in the body of the methods or in event handlers, which you can create on the Events page of the Inspector.

To set a component's properties at design time,

- 1 Select a component.

Any component can be selected in the component tree. Visible components can be selected on the design surface as well.

- 2 Click the Properties tab of the Inspector.

- 3 Select the property you want to change, using the mouse or arrow keys. You may need to scroll down until the property you want is visible.

- 4 Enter the value in the right column in one of the following ways:

- When there is a text field, type in the value for that property.
- When the value field has a drop-down list, click the arrow beside the property and choose a value from the list.

Either use the mouse or the *Up* and *Down* arrow keys on the keyboard to move through the list. Click or press *Enter* on the desired value.

- When the value field has an ellipsis (...) button, click the button to display a property editor, such as a color or font selector. Set the values in the property editor, then click OK or press *Enter*.

Rename a component in the Inspector by selecting its name in the value column and typing over the old name. When you rename a component, JBuilder performs a rename refactor, updating references to that component correctly.

## Setting shared properties

When more than one component is selected, the Inspector displays only the properties that

- The components have in common.
- Can be edited.

When you change any of the shared properties in the Inspector, the property value changes to the new value in all the selected components.

**Note** When the original value for the shared property differs among the selected components, the property value displayed in the Inspector is either the default or the value of the first component in the selection list.

To set properties for multiple components,

- 1 Select the multiple components that will have shared properties.
- 2 Select and edit the desired property in the Inspector.

## Setting a property when the drop-down list is empty

Sometimes the Inspector can't provide values for a property. To generate values,

- 1 Right-click the property in the Inspector.
- 2 Add objects of an appropriate type to the current class.

This populates the property value list. Use initialized objects for preference.

Normally, you can use the designer to add appropriate component objects.
- 3 Now you can select these objects as values from the property value list.

### See also

- [“Example” on page 43](#) in “Understanding property values”
- [“Setting property exposure” on page 41](#)

## Understanding the Inspector

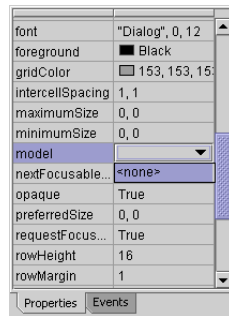
JBuilder's Inspector relies on information that's either contained in the BeanInfo class for the bean or derived from introspection of the bean itself. When no property editor is specified for a property in the bean's BeanInfo class, or if the bean does not have a BeanInfo class, the Inspector uses a default editor based on the property value's data type. For instance, if a property takes a `String`, the editor for that property lets you type in a string.

The list of property editors by type is stored in `propertyEditors2.properties` in the `<.jbuilder>` folder. If no default editor is registered for a particular data type, JBuilder builds a drop-down list containing all objects of the correct data type that are in scope. If there are no objects of the correct data type in scope, then the drop-down list is empty. You can create objects of the correct data type in scope and they will appear in the list.



### Example

For example, a `JTable` has a `model` property that takes objects of type `TableModel`. If you add a `JTable` to your design in the UI designer, then click the drop-down arrow on its `model` property in the Inspector, the drop-down list value is `<none>`.



**Note** One of the values the Inspector searches for is the property's exposure level. To make the `model` property visible in the Inspector, right-click in the Inspector and choose **Property Exposure Level** | **Hidden**.

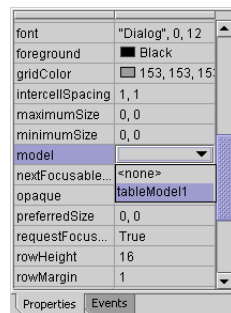
To populate the `model` property drop-down list in the Inspector, add objects of type `TableModel` to your class. For instance, you can add a `TableModel` class from the `javax.swing.table` package from the component palette:



- 1 Click the Bean Chooser button on the component palette.
- 2 Choose **Select**.
- 3 Expand the package `javax.swing.table`.
- 4 Select `TableModel`.
- 5 Click **OK**.
- 6 Click anywhere in the component tree.

This creates `tableModel1`.

Now the `model` property drop-down list for `JTable` in the Inspector is populated with `tableModel1`.



**Note** In some cases, you may need to add the object to your class manually, using the source code editor.



## Handling events

This chapter describes how to add, edit, and delete event handlers for components using the Inspector. It also gives specific examples of how to code commonly used event handlers for the JBuilder dialog box components.

*Event-handling code* is executed when the user interacts with the UI, as when clicking a button or choosing a menu item. Every component can be interacted with by the user. The component issues a message when a user interaction with that component occurs. In order to react to that interaction, the program must listen for the component's message and respond appropriately. The program needs a *listener* to listen for the component message, and an *event handler* to respond.

The Inspector's Events page lists all supported events for the selected component. Each event has a default action, out of several possible actions. When you double-click an event in the Inspector, JBuilder writes a listener and a *stub* (empty) event-handling method for the event's default action, and switches to the Source view with your cursor in the stub event-handler. Manually fill in the code describing what the program should do in response to that event.

There are some visual components, such as dialog boxes, that normally appear only when event-handling code is executed. (These components appear in the Default designer.) For example, a dialog box isn't part of the UI surface, but it's a separate UI element which appears transiently as a result of a user operation such as a menu choice or a button press. Therefore, some of the code associated with using the dialog, such as a call to its `show()` method, has to be placed into the event-handling method. This code is completely custom.

### See also

- [Chapter 4, "Using the component tree and Inspector"](#)
- ["Requirements for a class to be visually designable" on page 2](#)

## Attaching event-handling code

---

Using the Events page of the Inspector, you can attach event handlers to component events and delete existing handlers.

To attach event-handling code to a component event,

- 1 Select the component in the component tree or on the design surface.
- 2 Select the Events tab in the Inspector to display the events for that component.
- 3 Select an event. Use the mouse button or the arrow keys.
- 4 Double-click the event's name or press *Enter*.

JBuilder creates an event handler with an editable default name and switches to that event handler in the source code.

JBuilder also inserts code into your class, called an `Adapter`, to connect the event and the event-handling method.

- 5 Write the code inside the body of the event handler that specifies how you want the program to respond to that component event.

**Note** To find out what methods and events a component supports, view the documentation for that class. To do so, double-click that component in the component tree to load the class into the AppBrowser, tIDE the Doc tab.

### See also

- [“Connecting controls and events” on page 47](#)

## Creating a default event handler

---

To quickly create an event handler for a component's default event,

- 1 Select a component on the component palette and add it to your UI.
- 2 Double-click the component in the designer.

An event stub is created and focus switches to that event handler in the source code.

- 3 Add the necessary code to the event handler to complete it.

**Note** The default event is defined by `BeanInfo`, or as `actionPerformed` if none was specified.

## Deleting event handlers

---

To delete an existing event handler,

- 1 Select the component in the component tree or on the design surface.
- 2 Select the Events tab in the Inspector.
- 3 Click the event you want to delete.
- 4 Highlight the entire name of the event handler in the event's value field.
- 5 Press *Delete*.
- 6 Press *Enter* to remove the event handler name.

JBuilder deletes the hook to the associated event-handling method. If the handler is otherwise empty, JBuilder also deletes the adapter class from the source code. Delete the method itself manually.

## Connecting controls and events

---

Event adapters connect event handlers to their controls. You can use either standard event adapters or anonymous inner class adapters to accomplish this.

Standard adapters create a named class. The advantage to that is that the adapter is reusable, and can be referred to later and from elsewhere in the code. Anonymous adapters create inline code. The advantage to that is that the code is leaner and more elegant. However, it's single-use only.

When you use an anonymous adapter, the only code JBuilder creates is the listener and the event-handling stub. When you use a standard event adapter, JBuilder generates three pieces of code:

- The event-handling stub.
- An `EventAdapter`.
- An `EventListener`.

JBuilder creates an `EventAdapter` class for each specific component/event connection and gives it a name which corresponds to that particular component and event. This code is added in a new class declaration at the bottom of your file.

For example, in the `TextEdit` application, JBuilder generates a type of event adapter called an `ActionAdapter` with the following code:

```
// Creates the listener side of the connection:
class TextEditFrame_jMenuFileExit_ActionAdapter implements ActionListener {
    TextEditFrame adaptee;

    // Connects the adapter to the class:
    TextEditFrame_jMenuFileExit_ActionAdapter(TextEditFrame adaptee) {
        this.adaptee = adaptee;
    }

    // Provides the necessary ActionPerformed:
    public void actionPerformed(ActionEvent e) {
        adaptee.jMenuFileExit_actionPerformed(e);
    }
}
```

JBuilder also creates a line of code in the `jbInit()` method. This line of code connects the component's event source, through the `EventAdapter`, to your event-handling method. It does so by adding a listener. The listener method takes a parameter of the matching `EventAdapter`.

In the above example, the `EventAdapter` is constructed in place. Its constructor parameter is the `this` reference to `Frame` that contains the event-handling method. For example, here is the line of code that performs this task in the `HelloWorld` application:

```
jButton1.addActionListener(new Frame1_jButton1_actionAdapter(this));
```

The adapter class name is arbitrary. All that matters is that the reference matches.

JBuilder creates the adapter class with the implementation for the method in the `ActionListener` interface. The method that handles the selected event calls another method in the adaptee (`Frame1`) to perform the desired action.

## Standard event adapters

---

JBuilder generates an event adapter class that implements the appropriate interface. It then instantiates the class in the UI file and registers it as a listener for the component. For example, for a `JButton1` event, it calls `JButton1.addActionListener()`. All this code is visible in the source code. All that's left for you to do is to fill in the event-handling method that the action adapter calls when the event occurs.

For example, here is code that is generated for a `focusGained()` event:

```
JButton1.addFocusListener(new Frame1_JButton1_focusAdapter(this));

void jButton1_focusGained(FocusEvent e) {
    // code to respond to event goes here
}

class Frame1_JButton1_focusAdapter extends java.awt.event.FocusAdapter {
    Frame1 adaptee;

    Frame1_JButton1_focusAdapter(Frame1 adaptee) {
        this.adaptee = adaptee;
    }

    public void focusGained(FocusEvent e) {
        adaptee.jButton1_focusGained(e);
    }
}
```

The advantage to this adapter is that it can be reused, because it is named. The disadvantage is that it has only public and package access, which can impose limits on its usefulness.

## Anonymous inner class adapters

---

JBuilder can also generate inner class event adapters. Inner classes have the following advantages:

- The code is generated inline, thereby simplifying the appearance of the code.
- The inner class has access to all variables in scope where it is declared, unlike the standard event adapters that have only public and package access.

The particular type of inner class event adapters that JBuilder generates are known as *anonymous adapters*. This style of adapter creates a nameless adapter class. The advantage is that the resulting code is compact and elegant. The disadvantage is that this adapter can only be used for this one event, because it has no name and therefore cannot be called from elsewhere.

For example, the following is code generated for a `focusGained()` event using an anonymous adapter:

```
JButton1.addFocusListener(new java.awt.event.FocusAdapter() {
    public void focusGained(FocusEvent e) {
        jButton1_focusGained(e);
    }
})

void jButton1_focusGained(FocusEvent e) {
}
```

Compare this code with the standard adapter code sample shown above. JBuilder generated that code using a standard adapter class. Both ways of using adapters provide the code to handle `focusGained()` events, but the anonymous adapter approach is more compact.

## Choosing event handler style

---

When you create an event using the Events tab in the Inspector, JBuilder generates event adapter code in your container class to handle event listening. JBuilder lets you choose which style of event handling code is automatically generated. The two styles of event adapters are:

- Standard adapters
- Anonymous inner class adapters

To specify the style of event adapters JBuilder generates,

- 1 Choose Project|Project Properties.
- 2 Select the Formatting page.
- 3 Select the Generated tab inside the Formatting page.
- 4 Under Event Handling options, choose either Standard Adapter or Anonymous Adapter.

If you want the generated code to match existing event handlers, check the Match Existing Code option.

- 5 Click OK.

To choose a default event adapter style for all new projects,

- 1 Choose Project|Default Project Properties.  
This opens the Project Properties dialog box.
- 2 Choose Java Formatting|Generated.
- 3 Under Event Handling options, choose either Standard Adapter or Anonymous Adapter.

If you want the generated event handling code to match existing event handlers, check the Match Existing Code option.

- 4 Click OK.

## Examples: connecting and handling events

---

These are specific examples of frequently used event handlers:

- Displaying text when a button is pressed
- Invoking a dialog box from a menu item

### Displaying text when a button is pressed

---

Here is a simple example of connecting code that displays “Hello World!” in response to a button event:

- 1 Run the Application wizard to start a new application. Select File|New|General and double-click on Application.
- 2 Accept all the defaults and press Finish.

`Frame1.java` opens in the editor.

- 3 Click the Design tab at the bottom of `Frame1.java` to display the UI designer.
- 4 Select the `JTextField` and `JButton` components on the Swing page of the component palette.
- 5 Drop them in the component tree or on the design surface.
- 6 Select `jButton1` to display it in the Inspector.
- 7 Select the Events page in the Inspector.
- 8 Double-click the `actionPerformed` event.

You will be taken to the newly-generated stub in the source code of the event-handling method.

- 9 Enter the following code inside the braces of the event-handling method:

```
jTextField1.setText("Hello World!");
```

- 10 Run the application to try it out.



Click the Run button on the toolbar, or press *F9*.

When your application appears, click the button to see the text “Hello World!” appear in the text field. In this example, the `JFrame` component listens for the `actionPerformed` event on the button. When that event occurs, the `JFrame` sets the text in the `JTextField`.

### Example: Invoking a dialog box from a menu item

When you design your own programs, you will typically need to fill in the event-handling stubs. For example, you might want to extract the file name the user entered from a `JFileChooser` dialog box and use it to open or manipulate a file.

The following example shows you how to invoke a `JFileChooser` dialog box from a File|Open menu item:

- 1 Run the Application wizard in a new or existing project.
- 2 Press next on Step 1 to accept the defaults from this step.
- 3 Check Generate Menu Bar on step 2, then press Finish to accept the rest of the defaults.
- 4 Select the `Frame` file (`Frame1.java`) in the project pane and click the Design tab at the bottom of the AppBrowser to open the visual design tools.
- 5 Select the `JFileChooser` component on the Swing Containers page of the palette, and click the `UI` folder in the component tree. A component called `jFileChooser1` is added to the `UI` section of the tree.

#### Note

If you drop this component anywhere else in the tree or the designer, it becomes a subcomponent of `this` rather than of the `UI` as a whole. It consequently becomes the `UI` for your `Frame1.java` file.

- 6 Create an Open menu item on the File menu as follows: (The example code below was generated with Open as `jMenuItem1`.)
  - a Select `MenuBar1` in the component tree and press *Enter* to open the Menu designer.
  - b Place the cursor on the File|Exit menu item in the designer and press the Insert Item button on the Menu designer toolbar. A new empty menu item is added.
  - c Enter `Open` as a new menu item.
- 7 Select the Open menu item in the designer or on the component tree, then click the Events tab in the Inspector.





- 8 Double-click the `actionPerformed` event in the Inspector to generate the following event-handling method stub in the source code:

```
void jMenuItem1_actionPerformed(ActionEvent e) {  
}
```

JBUILDER takes you to this event-handling method in the source code.

- 9 Inside the braces of the `actionPerformed` event-handling method, type the following:

```
jFileChooser1.showOpenDialog(this);
```

- 10 Save your files.

Now, run your program and use its `File|Open` menu commands.

### See also

- “Tutorial: Building a Java text editor” in online help



## Creating user interfaces

Creating a good user interface requires more than good programming. The important concerns of usability and the principles of good design are well-addressed in the many excellent books available on the subject. This chapter focuses on using JBuilder's tools to facilitate the process of implementing a user interface design.

This involves certain tasks:

- Creating a project that contains a main UI container, such as a `Frame`, `Panel`, or `Dialog`.
- Adding components to the UI container, such as additional containers, UI controls, and database components.
- Setting component properties.
- Attaching code to component events, so the UI can respond to user input.
- Setting container layouts and component constraints, so the UI can look and behave right after being deployed.

JBuilder assists by providing wizards to create appropriate frameworks of files for your project and visual design tools to speed up the UI design process.

**Note** Where visual design tools are the same across different types of designers, they're described in other sections. This chapter focuses on functionality specific to the UI designer.

### See also

- "Using the Application wizard" in online help, which creates a basic group of UI files and widgets
- ["Adding components" on page 36](#) to learn about adding components to your design and then adjusting them
- ["Setting property values" on page 41](#) to change the property values of beans
- ["Attaching event-handling code" on page 46](#) to use the Inspector to attach and edit event-handling stubs
- [Chapter 9, "Using layout managers"](#) for information on using Java's layout managers to make your UI look the way you want it to and behave correctly when resized

## Selecting components in the UI

---

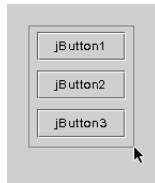
You have already created the basic structure of your UI, adding containers and a few other components to the design. You are ready to work with the basic UI components now in your design.



Before selecting existing components, be sure the Selection Arrow button on the component palette appears depressed. Otherwise you may accidentally place a new component on your design.

To select multiple components on the design surface, do one of the following:

- Hold down the *Ctrl* key and click the components on the design surface one at a time.
- Hold down the *Shift* key and drag around the outside of the components on the design surface, surrounding the components with a rectangle.



When this rectangle encloses all the components you want to select, release the mouse button. If necessary, you can then use *Ctrl+click* to individually add or remove components from the selected group.

### See also

- [“Adding components” on page 36](#) for more on how to add single and multiple components
- [“Action shortcuts” on page 33](#) in “Keyboarding in the designer”

## Adding to nested containers

---

Building even a moderately complex UI often involves nesting containers within other containers. For example, you might want to add a panel to a `BorderLayout` container that already contains two other panels. You need a way to indicate to the designer which container should actually receive the selected component.

To do this,

- 1 Select the container to which you want to add the component.  
Either select it in the component tree or select it in the designer, using the status bar to check that you’re choosing the correct container.
- 2 Select the component on the palette that you want to add.
- 3 Drop the component into the parent container on the design surface, continuing to hold the mouse button down.
- 4 Press the *Alt* key, and while holding it down, release the mouse button.

Another way to add a component to a container easily in a nested layout is to drop it on the target container in the structure pane’s component tree.

Once the component is in the new container, you can modify its constraints to specify its exact placement.

## Moving and resizing components

For many layouts, the layout manager completely determines the size of the components by constraints, so you can't size the components yourself. However, when the `layout` property is set to `null` or `XYLayout` in the Inspector, you can either size components when you first place them in your UI or resize and move them later.

To size a component as you add it,

- 1 Select the component on the component palette.
- 2 Place the cursor where you want the component to appear in the design.
- 3 Drag the mouse pointer before releasing the mouse button.

As you drag, an outline appears to indicate the size and position of the control.

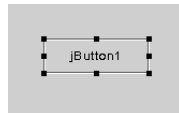


- 4 Release the mouse button when the outline is the size you want.

To resize a component, either edit its constraints in the Inspector or use the mouse:

- 1 Click the component on the design surface or in the component tree to select it.

When a component is selected, small squares, called *nibs* or *sizing handles*, appear on the perimeter of the component. For some containers, an additional nib called a *move handle* appears in the middle of the component.



- 2 Click an outer handle and drag to resize.

To move a component using the mouse,

- 1 Click the component on the design surface or in the component tree to select it.
- 2 Do one of the following on the design surface:
  - Click anywhere inside the component, and drag it any direction. If the component is a container completely covered with other components, use the center move handle to drag it.
  - Hold down the `Ctrl` key, and use one of the arrow keys to move the component one pixel in the direction of the arrow.
  - Hold down the `Shift+Ctrl` keys, and use one of the arrow keys to move the component eight pixels in the direction of the arrow.

To move a component without using the mouse,

- 1 Select the component in the component tree.
- 2 Cut the component. Either choose `Edit|Cut` or use the keyboard shortcut for your editor emulation.
- 3 Select the parent container in the component tree.
- 4 Paste the component. Either choose `Edit|Paste` or use your keyboard shortcut.

**Important** If you don't use the mouse at all, be sure to add the components in the sequence which the final design will require them to appear in. If a component is added out of intended sequence, you must delete the subsequent components that are at or below its hierarchical level and rework the design from the corrected component onwards.

**See also**

- [“Cutting, copying, and pasting components” on page 37](#) to learn how to move and size components without using the mouse

## Managing the design

---

You have a project populated with visually designable components which have properties and events attached to them. Before the UI gets unwieldy, you want to group your components to make the UI as clean-looking and usable as possible. You'll need to make sure the application provides enough usefulness to the user. Once the UI is fully populated and its appearance and behavior are under control, you want to tune the look and feel and test the UI to make sure it looks and behaves as intended.

The rest of this chapter deals with these more advanced topics.

### Grouping components

---

Some components on the palette are containers that can be used to group components together so they behave as a single component at design time.

For example, you might group a row of buttons in a `Panel` to create a toolbar. Or you could use a container component to create a customized backdrop, status bar, or check box group.

When you place components within containers, you create a relationship between the container and the components it contains. All design-time operations you perform on the containers, such as moving, copying, or deleting, also affect any components grouped within them.

To group components by placing them into a container,

- 1 Add a container to the UI.

If you are working in `null` or `XYLayout`, you can drag to size it.

- 2 Add each component to the container:

- Via mouse, make sure the mouse pointer falls within the container's boundaries.
- Via keyboard, put each component within the container in the component tree.

You can drop a new component from the component palette, or drag an existing component into the new container.

The status bar at the bottom of the IDE displays which container your cursor is over in the design surface.

As you add components, they appear inside the selected container on the design surface and they appear under that container in the component tree.

**Tip** If you want the components to stay where you put them, change the container's layout to `null` or `XYLayout` before adding any components. Otherwise, the size and position of the components will change according to the layout manager used by the container. You can change to a final layout after you finish adding the components.

### Adding application-building components

---

Components that do not descend from `java.awt.Component`, such as menus, dialog boxes and database components, are treated differently from UI components during class design. They are represented on the component palette, but when you add them to your class, they are visible only in the component tree. They also use different designers. You can select them in the component tree to change their properties in the Inspector or double-click them to open the associated designer.

## Menus

To add menus to your UI,

- 1 Click one of the following menu bar or context menu components on the component palette, or select from Edit/Add Components:

|                       |                        |
|-----------------------|------------------------|
| Swing Containers page | JMenuBar<br>JPopupMenu |
| AWT page              | MenuBar<br>PopupMenu   |

- 2 Drop it anywhere on the component tree or on the design surface. Notice that it is placed in the component tree's `Menu` folder.
- 3 Double-click the menu component in the component tree to open the Menu designer, or right-click it and choose `Activate Designer`.
- 4 Add menu items in the Menu designer.
- 5 Attach events to the menu items by using the Inspector or manually typing the code.
- 6 Close the Menu designer by double-clicking a UI component in the component tree.

### See also

- [Chapter 7, “Designing menus”](#)

## Dialog boxes

There are two ways to add dialog boxes to your project automatically:

- Use an existing one from the component palette.
- Create a custom one using the Dialog wizard in the object gallery (`FileNew`).

To add an existing dialog,

- 1 Select a dialog component, such as `JFileChooser`, from the component palette. You'll find them on the Swing Containers page and on the More dbSwing page.
- 2 Drop it on the UI folder in the component tree.

**Note** Depending on the type of dialog, it is placed in either the UI folder or Default folder of the component tree.

- 3 Attach events to the associated menu item that will surface the dialog at runtime. Use the Events tab in the Inspector or create the source code manually.

**Tip** If you're using dbSwing components, select them in the component tree and change the `frame` property to `this` so they're visible at runtime.

To create a custom dialog with the Dialog wizard,

- 1 Choose `FileNew` and open the Dialog wizard icon in the object gallery.
- 2 Name your dialog class, and choose the base class from which you want the class to inherit.
- 3 Click OK to close the dialog box.

A shell dialog class is created in your project with a `Panel` added so it is ready to design in the designer.

- 4 Complete any UI design desired, then attach events to the menu items that will surface the dialog at runtime.

For information on how to hook up menu events to dialog boxes, see [Chapter 5, “Handling events.”](#)

Once the dialog box has been created and its UI designed, you will want to test or use your dialog box from some UI in your program.

To use a dialog that is not a bean,

- 1 Instantiate your dialog class from someplace in your code where you have access to a `Frame` which can serve as the parent `Frame` parameter in the dialog constructor. A typical example of this would be a `Frame` whose UI you are designing, that contains a button or a menu item which is intended to bring up the dialog. In applets, you can get the `Frame` by calling `getParent()` on the applet.

For a modeless dialog box (which we are calling `dialog1` in this example), you can use the form of the constructor that takes a single parameter (the parent `Frame`) as follows:

```
Dialog1 dialog1=new Dialog1(this);
```

For a modal dialog box, you will need to use a form of the constructor that has the `boolean modal` parameter set to `true`, such as in the following example:

```
Dialog1 dialog1=new Dialog1(this, true);
```

You can either place this line as an instance variable at the top of the class (in which case the dialog box will be instantiated during the construction of your `Frame` and be reusable), or you can place this line of code in the `actionPerformed()` event handler for the button that invokes the dialog box (in which case a new instance of the dialog will be instantiated each time the button is pressed.) Either way, this line instantiates the dialog, but does not make it visible yet.

In the case where the dialog box is a bean, you must set its `frame` property to the parent frame before calling `show()`, rather than supplying the frame to the constructor.

- 2 Before making the instantiated dialog box visible, you should set up any default values that the dialog fields should display. If you are planning to make your dialog box into a Bean (see below), you need to make these dialog box fields accessible as properties. You do this by defining getter and setter methods in your dialog class.
- 3 Next, you have to cause the dialog box to become visible during the `actionPerformed()` event by entering a line of code inside the event handler that looks like this:

```
dialog1.show();
```

- 4 When the user presses the OK button (or the Apply button on a modeless dialog box), the code that is using the dialog box will need to call the dialog's property getters to read the user-entered information out of the dialog, then do something with that information.
  - For a modal dialog box, you can do this right after the `show()` method call, because `show()` doesn't return until the modal dialog is dismissed.
  - For a modeless dialog, `show()` returns immediately. Because of this, the dialog class itself will need to expose events for each of the button presses. When using the dialog box, you will need to register listeners to the dialog's events, and place code in the event handling methods to use property getters to get the information out of the dialog box.

### See also

- "Tutorial: Building a Java text editor" in online help to see examples of using modal dialog box components



## Database components

Database components are JavaBean components that control data and are often attached to data-aware UI components. They often don't show in the UI themselves. They're located on the DataExpress page of the component palette.

To add a database component to your class using your mouse,

- 1 Select the DataExpress page of the component palette and click the desired component.
- 2 Drop it on the component tree or on the design surface.

Although the component is not visible in the UI designer, it appears in the Data Access designer's folder in the component tree.

- 3 Modify any properties and add event handlers as with other components.

To add a database component using menus,

- 1 Choose Edit|Add Component.  
The Add Component dialog appears.
- 2 Select a component from the DataExpress library.
- 3 Click OK or press *Enter*.

To use the column designer,

- 1 Add a component with columns, such as a `TableDataSet` or a `QueryDataSet`.  
Use either technique above.
- 2 Click the expand icon beside the component in the component tree.
- 3 Double-click the <newcolumn> node to open the column designer.  
Use the column designer to adjust the appearance and behavior of the column.
- 4 Close the column designer by double-clicking any non-database component in the component tree.

### See also

- "Understanding JBuilder database applications" in *Developing Database Applications* for complete information about using Database components

## Changing look and feel

---

In multi-platform development, a design must appear and behave predictably on each platform the program is expected to support. Most developers work primarily on one platform. It's hard to predict how a UI will look and feel on platforms you're not extremely familiar with.

JBuilder provides several straightforward ways of changing the look and feel both at run time and at design time.

### Runtime look and feel

JBuilder includes the Java Foundation Classes (Swing) graphical user interface (GUI) components. Swing architecture gives you the capability of specifying a look and feel for a program's user interface. You can take advantage of this Java feature to create applications that will have the look and feel of a user's native desktop. You can also ensure a uniform look and feel in your applications across platforms with the Java Metal Look & Feel.

There are several choices of look and feel available to you in JBuilder:

- Borland
- Metal
- CDE/Motif
- Windows (supported only on Windows platforms)
- MacOS Adaptive (supported only on Macintosh platforms)

When you create an application or an applet using the JBuilder wizards, the following code is automatically generated in the class, `Application1.java` or `Applet1.java` for example, that runs your program.

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

For example,

**Application:**

```
//Main method
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
    }
    new Application1();
}
```

**Applet (when base class is `javax.swing.JApplet`):**

```
//static initializer for setting look and feel
static {
    try {
        //UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        //
        UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
    }
    catch(Exception e) {
    }
}
```

JBuilder uses the following method to set the program's look and feel:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

This method automatically detects which platform is running your program and uses that look and feel for your program.

Note that the line of code for the look and feel statement is inside the `try/catch` block. The `setLookAndFeel()` method throws a number of exceptions that need to be explicitly caught and handled.

You can change the runtime look and feel to Metal or Motif by changing the code in the `UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());` method to one of the following:

```
// Metal look and feel:
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

// Motif look and feel:
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

**Important** If you know you want your runtime look and feel to be Motif, then be sure to use Motif in the designer so you can see the end results. Motif puts more space around some components, such as buttons.

## Design time look and feel

The runtime look and feel is determined by the source code setting. The look and feel selected in JBuilder's designer is for preview purposes and has no effect on the source code. You can change the look and feel in the designer at any time to preview how the design looks in a particular look and feel, and later change the code to set that look and feel if you like it.

There are two ways to change the design time look and feel:

- Right-click on the design surface and choose Look And Feel. Select an alternate look and feel from this submenu.

This only changes the look and feel in the designer and only for the current project. Using this method, you can design in one look and preview it in the runtime look, all from within the designer.

- Choose Tools|IDE Options. Select an alternate look and feel from the Look And Feel drop-down list on the Browser page.

This changes the look and feel for the JBuilder environment, but you can still use the first method to switch the look in the designer.

The design surface repaints to display the selected look and feel. It does not change your code.

### Important

Changing the look and feel in the designer does not change your code. This is only a preview in the designer and does not affect the look and feel at runtime.

## Testing the UI at runtime

---

When you're ready to test your program, you can simply run it, or you can run it and debug it at the same time.



To run your application, choose Run|Run Project, press *F9*, or click the Run button on the toolbar. JBuilder compiles the program, and if there are errors, compiling stops so you can fix the errors and try again.



To debug your application, choose Run|Debug, press *Shift+F9*, or click the Debug button. Debug your program and correct any errors.

For complete information on these topics see the following chapters in *Building Applications with JBuilder*.

### See also

In *Building Applications with JBuilder*:

- "Running Java programs"
- "Building Java programs"
- "Compiling Java programs"
- "Debugging Java programs"
- "Deploying Java programs"



## Designing menus

This chapter shows you how to visually design menus. Using the JBuilder Menu designer, you can visually design both menu bar menus and pop-up menus.

### Opening the Menu designer

---

To open the Menu designer

- 1 Click the Design tab in the content pane to open the UI designer.

Once the designer is open, either

- Click the Menu tab at the bottom of the content pane, or
- Expand the Menu folder in the component tree and double-click any menu component in the component tree.

The design surface changes to show the Menu designer features.

If there are no components in the Menu folder, you must first bring a menu component into the design.

- 2 Use either the Add Component dialog or the component palette to add one of the following menu bar or pop-up menu components:

- From the Swing Containers palette:

JMenuBar  
JPopupMenu

- From the AWT palette:

MenuBar  
PopupMenu

- 3 Drop the component onto the component tree or into the design surface.

It is automatically placed in the component tree's Menu folder.

Toggle between the Menu designer and any other designer type in one of these ways:

- Click the designer type's tab, within the Design tab in the content pane.
- Double-click a component of the desired designer type in the component tree.
- Select a component of the desired designer type in the component tree and press *Enter*.
- Right-click a component of the desired designer type in the component tree and choose *Activate Designer*.

As you edit menu items in the Menu designer, all changes are immediately reflected in the Inspector, the component tree, and the source code. Likewise, when you make changes in the source code, the changes are reflected in the IDE.

There is no need to save your menu design explicitly. Code is generated as you work in the Menu designer and is saved when you save your `.java` source file. The next time you open the `.java` file and click a `MenuBar` component in the component tree, the Menu designer will open and reload everything for that component.

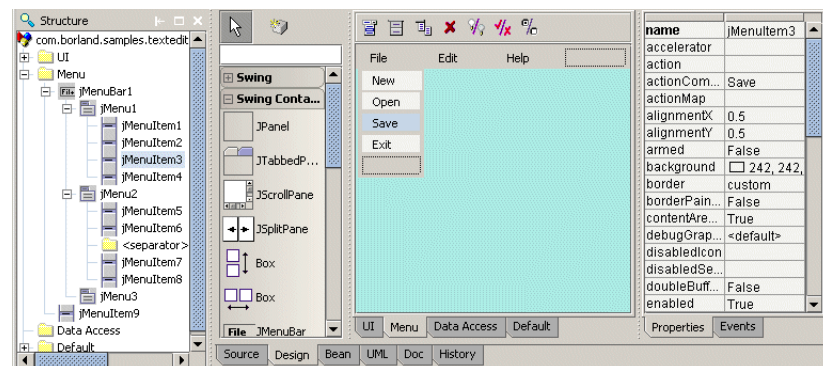
## Menu terminology

The basic parts of a menu are referred to using the following terms:

- A *menu* is a list of choices, or menu items, which the user interacts with at runtime.
- A *menu item* is one choice on a menu. Menu items can have attributes such as being disabled (gray) when not available to the user, or checkable so their selection state can be toggled.
- The *menu bar* is located at the top of a frame and is composed of menus containing individual menu items.
- A *submenu* is a nested menu accessed by clicking on an arrow to the right of a menu item.
- A *keyboard shortcut* is displayed to the right of the menu item, and may be specific to a particular editor interface. For example, *Ctrl+X* is used to indicate *Edit/Cut* in many editors.
- The *separator* is a line across the menu which visually separates different groups of menu items.








## Menu design tools

When you open the Menu designer on a menu item, the design surface changes to look something like this:



This Menu designer has its own toolbar and recognizes keystrokes such as the navigation arrows and *Ins* and *Del* keys.

The Menu designer toolbar contains the following tools:

| Tool                                                                              | Action                                                                                                                                                                     |
|-----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | Inserts a placeholder for a new menu to the left of the selected menu or a new menu item above the selected menu item.                                                     |
|  | Inserts a separator immediately above the currently selected menu item.                                                                                                    |
|  | Creates a placeholder for a nested submenu and adds an arrow to the right of the selected menu item.                                                                       |
|  | Deletes the selected menu item (and all its submenu items, if any).                                                                                                        |
|  | Toggles the <code>enabled</code> property of the selected menu item between <code>true</code> and <code>false</code> and dims it when it is <code>false</code> (disabled). |
|  | Makes the menu item checkable.                                                                                                                                             |
|  | Toggles the menu item between being a <code>JMenuItem</code> or a <code>JRadioButtonMenuItem</code> .                                                                      |

**Note** Right-click a menu item in the Menu designer to display a pop-up menu containing many of the same commands.

## Creating menus

A menu bar must be attached to a `Frame` or a `JFrame` container. To create a menu in your application, first create a frame component file. Do this in one of the following ways:

- Create a new application with the Application wizard. On Step 2 of the wizard, check **Generate Menu Bar**.
- Open an existing frame component's file.
- Use the Frame wizard to add a `Frame` file to your project.

To add a menu component to the UI,

- 1 Select the `Frame` or `JFrame` file in the project pane.
- 2 Click the **Design** tab at the bottom of the **AppBrowser**.
- 3 Select your main UI frame on the design surface or in the component tree.
- 4 Click the menu component you want from either the **AWT** page or the **Swing Containers** page of the component palette.
- 5 Drop it anywhere on the design surface or in the component tree.
  - A `MenuBar` or `JMenuBar` is attached to the main UI `Frame`, and appears at the top of the application at runtime.
  - A `PopupMenu` or `JPopupMenu` appears when the user right-clicks in your UI. Pop-up menus do not have menu bars.

The menu component you added shows up as a node in the component tree, and its properties are displayed in the **Inspector**.

For every menu you want to include in your application, add a menu component to the target UI container. The first `MenuBar` component dropped onto the UI container is considered the current `MenuBar` for your UI. However, you can create more than one `MenuBar` for an application. Each `MenuBar`'s name is displayed in the Inspector in the frame's `MenuBar` property. To change the current `MenuBar`, select a menu from the `MenuBar` property drop-down list.

**Note** At design time, menu components are visible only in the Menu designer (not in the UI designer). However, you can always see them and select them in the component tree. To see how the menu looks in your UI, you must run your application.

## Adding menu items

---

When you first open the Menu designer, it displays the first blank menu item, indicated by a dotted rectangle.

- 1 Type a label for the menu item. (You may have to double-click inside the rectangle to get focus.)

The field is a default width if that menu list is empty. When it's not empty, the field is as wide as the longest menu item in the menu. While you're typing, the text field will scroll to accommodate labels longer than the edit field.

- 2 Press *Enter*, or press the *Down* arrow key to add another menu item.

A placeholder for the next menu or menu item is automatically added.

The name of the menu item in the component tree changes to reflect the label.

- 3 Type a label for each new item you want to create in the list, or press *Esc* to return to the menu bar.

You can use the arrow keys to move from the menu bar into a menu, and to move between items in the list.

## Inserting and deleting menus and menu items

---

To insert a new menu or menu item into an existing menu, select the rectangle where you want the new menu item to be (a new menu is inserted to the left of the selected menu on the menu bar, and a new menu item is inserted above the selected item in the menu list). Then do one of the following:



- Click the Insert button on the toolbar.
- Press the *Ins* key.
- Right-click and choose Insert Menu or Insert Menu Item.

To delete a menu item, select the menu item you want to delete, and do one of the following:



- Click the Delete Item button on the toolbar.
- Press the *Del* key.

**Note** A default placeholder (which you cannot delete) appears after the last menu on the menu bar and below the last item on a menu. This placeholder does not appear in your menu at runtime.



## Inserting separators

---

A separator inserts a horizontal line between menu items. You can use separators to group items within a menu list, or simply to provide a visual break in a list.

To insert a separator on a menu,

- 1 Select the menu item before which you want a separator.
- 2 Click the Insert Separator button on the toolbar.



The separator is inserted above the selected menu item.

## Specifying accelerator keys

---

Accelerator keys enable the user to perform a menu action by typing in a shortcut key combination. For example, a commonly used shortcut for File|Save is *Ctrl+S*.

To specify an accelerator key for a menu item,

- 1 Select the menu item in the Menu designer or in the component tree.
- 2 In the Inspector, select the `accelerator` property.
- 3 Either enter a value or choose a key combination from the drop-down list.

This list is only a subset of the valid combinations you can use.

When you add a shortcut, it appears next to the menu item at runtime. It does not appear in the designer.

## Disabling (dimming) menu items

---

You can prevent users from accessing certain menu commands based on a particular state of the current program, without removing the command from the menu. For example, if no text is currently selected in a document, the Cut, Copy, and Delete items on the Edit menu appear dimmed.

To disable a menu item, do one of the following:

- Click the Disable button on the toolbar.  
The Disable button toggles the `enabled` property for the selected menu item between `true` (enabled) and `false` (disabled).
- In the Inspector, set the `enabled` property for the menu item to `false`. (In contrast to the `visible` property, the `enabled` property leaves the item visible. A value of `false` simply dims the menu item.)

## Creating checkable menu items

---

To make a menu item checkable, you must change the menu item from a regular `JMenuItem` component to a `JCheckBoxMenuItem`. A `JCheckBoxMenuItem` has a `state` property (`boolean`) that determines if the associated event or behavior should be executed.

- A checked menu item has its `state` property set to `true`.
- An unchecked menu item has its `state` property set to `false`.

To change a regular menu item to a `JCheckBoxMenuItem`, either select the menu item and click the Check button on the toolbar or right-click the menu item and choose Make It Checkable.



## Creating Swing radio button menu items

---

The Menu designer lets you create Swing menu items that are part of a `ButtonGroup` in which only one item in the group can be selected. The selected item displays its selected state, causing any other selected items to switch to the unselected state.

To create a group of Swing radio button menu items,

- 1 In the Menu designer, create a menu or nested menu containing the menu items you want in the radio button group.
- 2 Right-click each of these menu items in the designer and choose **Toggle Radio Item**.



You could also select the menu item and click the `RadioButton` icon on the Menu designer toolbar.



- 3 Click the **Bean Chooser** button on the component palette to open the Package browser.

If you have already added packages to your Bean Chooser list, choose **Select**.

- 4 Expand `javax.Swing` and double-click `ButtonGroup`.

The selection cursor is now holding `ButtonGroup`.

- 5 Click anywhere in the tree or the design surface to add a `buttonGroup1` to the **Default** folder in the tree, and add the line `ButtonGroup buttonGroup1 = new ButtonGroup();` to the class variables.

- 6 Click the **Source** tab to open the editor.

- 7 In the constructor's **try** block after `jbInit()`, add each `JRadioButtonMenuItem` to `buttonGroup1`.

For instance,

```
//Construct the frame
public Frame1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
        jbInit();
        buttonGroup1.add(jRadioButtonMenuItem1);
        buttonGroup1.add(jRadioButtonMenuItem2);
        buttonGroup1.add(jRadioButtonMenuItem3);
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

- 8 Click the **Design** tab to open the UI designer.
- 9 Decide which radio button menu item you want selected as the default. In the **Inspector**, set its `selected` property to `true`.

The following line of code is added to your source:

```
jRadioButtonMenuItem2.setSelected(true);
```

- 10 When you run your program, you'll see the menu item whose `selected` property you set as `true` has a radio button to the left of it. When you click one of the other menu items, the radio button moves to the newly selected item.

## Moving menu items

---

In the Menu designer, you can move menus and menu items simply by dragging and dropping them with the mouse. When you move a menu or a submenu, any items contained in it move as well.

You can move the following:

- Menus along the menu bar.
- Menu items within a menu.
- Menu items to other menus.
- Entire menus to a nest under a different menu item. (These become submenus.)
- Submenus up to the menu bar to create new menus.
- Submenu items to other menus.

The only exception to this is hierarchical: you cannot move a menu from the menu bar into itself; nor can you move a menu item into its own submenu. However, you can move any menu item into a different menu regardless of its original position.

To move a menu or menu item using the mouse,

- 1 Drag it with the mouse until the tip of the cursor points to the new location.

To move the menu or menu item to another menu, drag it to the target menu. The target menu opens.

- 2 Release the mouse button to drop the menu item into the new location.

To move a menu or menu item using the keyboard, use the component tree.

### See also

- [“Moving components” on page 38](#) for more on using the component tree

## Creating submenus

---

Many applications have menus containing drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. JBuilder supports as many levels of such nested menus, or submenus, as you want to build into your menu. However, for optimal design purposes you probably want to use no more than two or three menu levels in your UI design.

When you move a menu off the menu bar into another menu, its items become a submenu. Similarly, if you move a menu item into an existing submenu, its sub-items then form another nested menu under the submenu.

You can move a menu item into an existing submenu, or you can create a placeholder at a nested level next to an existing item, and then drop the menu item into the placeholder to nest it.

To create a submenu,

- 1 Select the menu item that will have a submenu and do one of the following:



- Click the Insert Nested Submenu button on the toolbar.
- Right-click the menu item and choose Insert Submenu.

- 2 Type a name for the nested menu item, or drag an existing menu item into this placeholder.

- 3 Press *Enter*, or the *Down* arrow, to create the next placeholder.
- 4 Repeat steps 2 and 3 for each item you want to create in the nested menu.
- 5 Press *Esc* to return to the previous menu level.

## Moving existing menus to submenus

---

You can also create a submenu by inserting a menu from the menu bar between menu items in another menu. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact nested menu. This pertains to moving submenus as well; moving a submenu into another submenu just creates one more level of nesting.

## Attaching code to menu events

---

A menu item has only one event: `actionPerformed()`. Code that you add to the `actionPerformed()` event for a menu item is executed whenever the user chooses that menu item or uses its keyboard shortcut.

To add code to a menu item's `actionPerformed()` event,

- 1 In the Menu designer, select a menu item.
- 2 In the Inspector, select the Events tab.
- 3 Select the column beside `actionPerformed()` and double-click to create an event-handling method skeleton in the source code with a default name.

**Note** To override the default name for the `actionPerformed()` event-handling method, single click in the event's value field, type a new name for the event method and press *Enter*.

When you double-click the event value, the source code is displayed. The cursor is positioned in the body of the newly created `actionPerformed()` event-handling method, ready for you to type.

- 4 Inside the open and close braces, type the code you want to have executed when the user clicks this menu command.

### Example: Invoking a dialog box from a menu item

To display the File Open dialog box when the user chooses File|Open,

- 1 Create a File menu with an Open menu item.
- 2 On the Swing Containers page of the component palette, click the `JFileChooser` component and drop it on `UI` folder in the component tree.
- 3 In the Menu designer, or in the component tree, select the Open menu item.
- 4 In the Inspector, select the Events page.
- 5 Select the `actionPerformed()` event and double-click to generate the following event-handling method skeleton in the source code, with your cursor in the correct location, waiting for you to type:

```
void jMenuItem1_actionPerformed(ActionEvent e) {
    |
}
```

**Note** JBuilder takes you to the existing event-handling method if there is one.

- 6 Inside the body of the `actionPerformed()` method, type the following:

```
jFileChooser1.showOpenDialog(this);
```

- 7 In a real program, you will typically need to add several lines of custom code in the event-handling methods. For example, here you might want to extract the file name the user entered and use it to open or manipulate the file.

#### See also

- [“Attaching event-handling code” on page 46](#)
- [“Connecting controls and events” on page 47](#)

## Creating pop-up menus

---

To create a pop-up menu,

- 1 In the designer, click a `PopupMenu` component from the AWT page of the component palette or a `JPopupMenu` component from the Swing Containers page and drop it into the component tree.

It will be the selected item in the tree.

- 2 Press *Enter* on the selected `PopupMenu` or `JPopupMenu` in the component tree to open the Menu designer.

- 3 Add your items to the menu.

- 4 Select `this(BorderLayout)` in the component tree and press *Enter* to return to the designer.

- 5 Select the component having the event you want the pop-up menu attached to, so you can see that component in the Inspector.

For the example below, `jPanel1` was selected.

- 6 Click the Events tab in the Inspector.

- 7 Double-click the event you want to attach the pop-up menu to.

The `MouseClicked` event was selected in the example below.

- 8 Edit your event-handler skeleton code to:

- Add the `JPopupMenu` instance,
- Check for the user action (such as right-clicking) which will display the pop-up, and
- Make the menu appear in the appropriate place.

It should resemble the following:

```
void jPanel1_mouseClicked(MouseEvent e) {
    jPanel1.add(jPopupMenu1); // JPopupMenu must be added to the component
    whose event is chosen.
    // For this example event, we are checking for right-mouse click.
    if (e.getModifiers() == Event.META_MASK) {
        // Make the jPopupMenu visible relative to the current mouse position in
        the container.
        jPopupMenu1.show(jPanel1, e.getX(), e.getY());
    }
}
```

- 9 Add event handlers to the pop-up menu items as needed for your application.



## Advanced topics

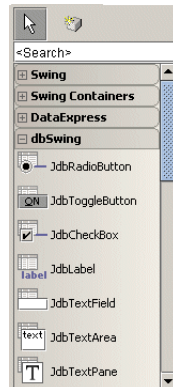
The component palette can be customized to include additional component libraries and individual JavaBeans. If a component doesn't meet specific criteria, it may show up as a red bean. Distributed applications commonly require serializing, using customizers, and handling resource bundle strings. This document describes these tasks in JBuilder.

### Managing the component palette

---

The component palette provides quick access to components on the `CLASSPATH`. By default, JBuilder displays all components on the `CLASSPATH`, sorted by libraries. For instance, the Swing page in the component palette displays components in the Swing library.

JBuilder comes with several component libraries. Choose from among these using the different pages of the component palette, the Add Component dialog box, or the Bean Chooser button. You can add components to the existing libraries or create new libraries for them.



You might want to install additional components delivered with JBuilder, components you created yourself, or third-party components. The following sections explain how to install additional components and pages on the palette, delete unused ones, and customize the palette.

#### See also

- “Working with libraries” in *Building Applications with JBuilder*

## Adding a component to the component palette

---

If your component is a JavaBean, you can add it to the component palette.

If the component is to be shared between projects, it should be added to a library that is on the `CLASSPATH` in each project where it will be used.

To place the component on the component palette,

- 1 Choose `Tools|Configure|Palette`, or right-click a palette page and choose `Properties`, to display the `Palette Properties` dialog box.
- 2 Click the `Pages` tab.
- 3 In the `Pages` column, either
  - Select the palette page on which you want your component to appear, or
  - Click the `Add` button to create a new page.
- 4 Click the `Add Components` tab. This is where you select the components you're adding.
- 5 Press the `Select Library` button to open the `Select A Different Library` dialog box.
- 6 Select an existing library of components from the list or create a new one by pressing `New` and using the `New Library` wizard.
- 7 Click `OK` to close the library dialog box.
- 8 Select the palette page where the components will go.
- 9 Choose a filtering option from the `Add Components` page.
  - `JavaBeans In Jar Manifest Only`: automatically adds JavaBeans defined in the library's JAR manifest to the selected page of the component palette.
  - `JavaBeans With BeanInfo Only`: displays a list of JavaBeans with `BeanInfo` in the `Browse For Class` dialog box when the `Add From Selected Library` button is pressed.
  - `JavaBeans Only`: displays a list of JavaBeans only in the `Browse For Class` dialog box when the `Add From Selected Library` button is pressed.
  - `No Filtering`: displays an unfiltered list of all classes in the `Browse For Class` dialog box when the `Add From Selected Library` button is pressed.
- 10 Click the `Add From Selected Library` button.
  - If the `JavaBeans In JAR Manifest Only` option is selected, the JavaBeans are added automatically to the selected page. Continue to the last step to make your changes.
  - If you selected one of the other options, the `Browse For Class` dialog box displays. Select the individual classes you want, then click `OK`. A `Results` dialog box displays with the classes listed. Click `OK`.
- 11 Click `OK` to close the `Palette Properties` dialog box and make your changes.

The JBuilder component palette manager is an `OpenTool` that lets you add your own classes to the palette. Remember to keep backend material (access to databases, socket connections to other computers, distributed object interactions using RMI or CORBA, calculations and computations) in classes other than the UI classes.



## Selecting an image for a component palette button

---

The image on a component palette button can be one of three things:

- An image defined in the bean's BeanInfo class.
- A .gif file you select from the Component Properties dialog box.
- A default image provided by JBuilder if neither of the above are provided.

To select the image for your component palette button,

- 1 Either choose Tools|Configure|Palette, or right-click the component palette and choose Properties, to open the Palette Properties dialog box.
- 2 Choose the Pages tab.
- 3 Select the appropriate library in the Page column and the component in the Components column.
- 4 Either double-click the component, or click the Properties button to open the Component Properties dialog box.

**Note** An image of the component's icon is displayed in the Component Properties dialog box to the left of the Icon options.

- 5 Do one of the following:
  - Choose Use JavaBean Icon to use the image provided by the bean for the button.
  - Choose Select Image and click the Browse button to choose a .gif file to be displayed on the button.
  - Type in a new or different initialization string if you want to use a factory method or a constructor with more than one parameter, instead of the default parameterless Bean constructor.

For best results, use a 32x32 .gif file.

- 6 Click OK to close the Component Properties dialog box.
- 7 Click OK in the Palette Properties dialog box when you're finished.

## Adding pages to the component palette

---

You can add a page of components to the component palette if you have the library of components on your path. To add a page to the palette,

- 1 Either choose Tools|Configure|Palette, or right-click the component palette and choose Properties.
- 2 Select the Pages tab in the Palette Properties dialog box.
- 3 Click the Add button to open the Add Page dialog box.
- 4 Enter a name for the page in the Page Name field.
- 5 Enter a description in the Page Description field.
- 6 Click OK.
- 7 Select the new page in the Pages column and click the Move Up button to move it to the desired location on the palette.

Pages added to the component palette also display in the Add Component dialog box.

## Removing a page or component from the component palette

---

To remove a page or component from the palette,

- 1 Either choose `Tools|Configure|Palette`, or right-click the component palette and choose `Properties`.
- 2 Select the `Pages` tab in the `Palette Properties` dialog box.
- 3 Select the appropriate page in the `Page` column and the component in the `Components` column.
- 4 Click `Remove`, then click `OK`.

Pages and components removed from the component palette no longer appear in the `Add Component` dialog box.

**Note** Removing a component from the component palette removes only the shortcut to that component. It does not remove the component from its library. Once a page or component has been removed from the component palette, it's still accessible from the `Browse` or `Search` page of the `Add Component` dialog box.

## Reorganizing the component palette

---

To change the order of the pages or components on the palette,

- 1 Choose `Tools|Configure|Palette`, or right-click the component palette and choose `Properties`.
- 2 Select the `Pages` tab.
- 3 Select a page in the `Pages` column or a component in the `Components` column.
- 4 Click either `Move Up` or `Move Down` to move the selected item to a new location.
- 5 Click `OK` when you are finished.

## Handling red beans

---

If JBuilder can't create an instance of a component for the designer, the component appears as a red box with its class name at the top. This is called a *red bean*. Red beans appear when:

- The class, or its constructor, is not `public`.
- The class threw exceptions on every constructor JBuilder tried.
- The class is the `this` object and extends an `abstract` class.

Fix the first two issues by supplying a `public` default constructor.

The third issue is more complex. When JBuilder needs to instantiate an object to be the `this` object, it can't instantiate the object you're designing. To circumvent this problem, JBuilder instantiates the superclass of the `this` object. If the superclass is `abstract`, it can't be instantiated. It's necessary to provide a concrete (non-abstract) proxy class. If this is what you need to do, please consult the Release Notes in the directory containing your JBuilder installation.

## Serializing

---

*Serializing* an object is the process of turning it into a sequence of bytes and saving it as a file on a disk or sending it over a network. When a request is made to restore an object from a file, or on the other end of a network connection, the sequence of bytes is *deserialized* into its original structure.

For JavaBeans, serialization provides a simple way to save the initial state for all instances of a type of class or bean. If the bean is serialized to a file and then deserialized the next time it's used, the bean is restored exactly as the user left it.

**Warning** Use extreme caution when serializing components. Don't attempt it until you know exactly what it entails and what the ramifications are.

### See also

- "Serialization" in *Getting Started with Java*
- Sun's article, "Object Serialization", at <http://java.sun.com/j2se/1.4.2/docs/guide/serialization/>

## Serializing components in JBuilder

---

JBuilder makes it easy to serialize JavaBeans in the designer:

- 1 Modify the bean using the Inspector to give it any settings you want to keep.
- 2 Select and right-click the component in the component tree.
- 3 Choose Serialize from the menu that appears.

A message box appears, showing you the path and name for the new serialized file. It has the same name as the original component and a `.ser` extension.

- 4 Click OK to create the serialized file.

The following occurs:

- A confirmation dialog box appears if the file already exists.
- The `.ser` serialization file is created starting at the first directory named on the Source Path, and it creates the appropriate package subdirectory path (for example `myprojects/java/awt`).
- The serialization file is added to your project as a node so it can be deployed with your project.
- JBuilder copies the `.ser` file from the Source Path to the Out Path during compiling.

The next time you instantiate that bean using `Beans.instantiate()`, the `.ser` file is read into memory.

## Serializing a this object

---

To serialize, you need a live instance of the object you want to serialize. This presents a problem for a `this` object, because it is not instantiated the same way as components added to the designer. Therefore, you need an alternate way to get a live instance of `this`.

Of course you can serialize in code, but you generally serialize an object after you have used an Inspector) or a customizer to make changes to it.

**Example**

The following example uses a `this` UI frame. You can modify these steps to serialize any type of object, such as a `panel`, `class`, `dialog`, or `menu`.

- 1 Open your project and create the class you want to serialize. (`Frame1.java`, for this example).
- 2 Save and compile it.
- 3 Open another file in your project that already has a class in it which can be, or has been, visually designed in JBuilder. (`OtherFile.java` in this example.)
- 4 Select `OtherFile.java` in the project pane, and create the following field declaration (instance variable) inside its class declaration, after any other declarations:

```
Frame1 f = new Frame1();
```

- 5 Click the Design tab to open the UI designer for `OtherFile.java`.
- 6 Right-click the `f` instance variable in the Default folder in the component tree and choose **Serialize**. A message box appears, indicating the path and file name for the new serialized file.
- 7 Click OK.

To use the serialized file when you instantiate `Frame1` in your application, you need to instantiate it in the Application file using `Beans.instantiate()`, as follows.

- 1 Change the constructor for `Application1.java` from

```
public Application1() {
    Frame1 frame = new Frame1();
    //Validate frames that have preset sizes
    //Pack frames that have useful preferred size info, e.g., from their layout
    if (packFrame)
        frame.pack();
    else
        frame.validate();
    frame.setVisible(true);
}
```

to

```
public Application1() {
    try {
        Frame1 frame = (Frame1)Beans.instantiate(getClass().getClassLoader(),
            Frame2.class.getName());
        //Validate frames that have preset sizes
        //Pack frames that have useful preferred size info, e.g., from their layout
        if (packFrame)
            frame.pack();
        else
            frame.validate();
        frame.setVisible(true);
    }
    catch (Exception x) {
    }
}
```

- 2 Add the following import statement to the top of `Application1.java`:

```
import java.beans.*;
```

## Using customizers in the designer

---

A JavaBean can name its own *customizer*, which is an editor specifically tailored for the bean's class. During design time in JBuilder, right-click the component in the component tree and choose Customizer; any JavaBean that has a customizer displays the customizer's dialog box. This dialog box is a wrapper around the customizer that lets you modify the object like you would in a property editor invoked from the Inspector.

The customizer doesn't replace the JBuilder Inspector. The JavaBean simply hides any properties it doesn't want the Inspector to display.

### Modifying beans with customizers

---

One way to persist the results of a customized bean is through serialization. Therefore, whenever you close the customizer's dialog, JBuilder will prompt you to save the object to a new `.ser` file or overwrite the old.

- If a bean is loaded with `Beans.instantiate()` in the designer and is then modified by the customizer, it can be saved again, either over the old name or into a new file.
- If a bean is loaded with `Beans.instantiate()` in the designer and is then modified by the Inspector, JBuilder still uses its old technique of writing Java source code for those property modifications. This means a customized, serialized bean can be further modified with the Inspector.

JBuilder attempts to generate code for any property changes the customizer makes. Depending on how closely the customizer follows the JavaBeans specification, this may or may not be sufficient. Some sophisticated customizers make changes to the object that cannot be represented in code, leaving serialization as the only way to save the changes.

#### See also

- [Chapter 2, "Creating JavaBeans with BeansExpress"](#)
- "Bean Customization" at <http://java.sun.com/docs/books/tutorial/javabeans/customization/index.html>

## Handling resource bundle strings

---

**This is a feature of JBuilder Developer and Enterprise.**

You can use the Inspector to remove a property's `String` value from a `ResourceBundle` file after you've resourced your project, or you can add a `String` value to a new or existing `ResourceBundle` file.

Right-click a property that takes a `String` value and choose ResourceBundle. The Localizable Property Setting dialog box appears.

### Removing from resource bundles

---

To remove a resourced `String` from a `ResourceBundle`,

- 1 Select the `ResourceBundle`'s filename from the list in the Localizable Property Setting dialog box.
- 2 Select Store Text As String Constant.

This removes a resourced `String` from its resource file.

## Adding to resource bundles

---

To add a `String` value to a new or existing `ResourceBundle`,

- 1 Select the option to Store Text In ResourceBundle For Localization.

When the project doesn't already have a `ResourceBundle`, selecting this option brings up the Create ResourceBundle dialog box.

When there is a `ResourceBundle`, it becomes the default. To change it, click New.

In either case, the Create ResourceBundle dialog box appears:

- a Select or edit the name of the ResourceBundle in the Create ResourceBundle dialog box.
- b Select the type of the ResourceBundle.
- c Click OK.

The new ResourceBundle appears in the Localizable Property Setting dialog box.

- 2 Tell JBuilder how to generate the key.

Choose from one of the following options:

- Generate Key From String Value: use the value of the string from which you accessed this dialog box.
- Generate Key From Component And Property Names: use the names of the component and property from which you accessed this dialog box, rather than the property's string value.

The key generated by the option appears in the Resource Key field.

- 3 Click OK.

**Important**

If you remove a resourced component from the UI, it's not automatically removed from its resource file. This avoids damage in case the key is used somewhere else in your code. You must open the resource file and manually remove the entry.

**See also**

- "Internationalizing programs with JBuilder" in *Building Applications with JBuilder*
- Resource Strings wizard (Wizards\Resource Strings)

## Using layout managers

A program written in Java may be deployed on multiple platforms. If you were to use standard UI design techniques, specifying absolute positions and sizes for your UI components, your UI won't be portable. What looks fine on your development system might be unusable on another platform. To solve this problem, Java provides a system of portable layout managers. You use these layout managers to specify rules and constraints for the layout of your UI in a way that will be portable.

Layout managers can provide the following advantages:

- Correctly positioned components that are independent of fonts, screen resolutions, and platform differences.
- Intelligent component placement for containers that are dynamically resized at runtime.
- Ease of translation. If a string increases in length after translation, the associated components stay properly aligned.

### About layout managers

---

A Java UI container (`java.awt.Container`) uses a special object called a *layout manager* to control how components are located and sized in the container each time it is displayed. A layout manager automatically arranges the components in a container according to a particular set of rules specific to that layout manager.

The layout manager sets the sizes and locations of the components based on various factors such as

- The layout manager's layout rules.
- The layout manager's property settings, if any.
- The layout constraints associated with each component.
- Certain properties common to all components, such as `preferredSize`, `minimumSize`, `maximumSize`, `alignmentX`, `alignmentY`.
- The size of the container.

Each layout manager has characteristic strengths and drawbacks. There are enough layout managers to choose from that you can find a layout manager to meet the requirements of each of your containers.

When you create a container in a Java program, you can either accept the default layout manager for that container type or override the default by specifying a different type of layout manager.

Normally, when coding a UI manually, you'll override the default layout manager before adding components to the container. When using the designer, you can change the layout whenever you like. JBuilder adjusts the code as needed on the fly. Change the layout either by explicitly adding a layout manager to the source code for the container, or by selecting a layout from the container's `layout` property list in the Inspector.

**Important** You cannot edit the `layout` properties for a `<default layout>`. If you want to modify the properties for a container's layout manager, you must specify an explicit layout manager; then its properties will be accessible in the Inspector.

## Using null and XYLayout

---

You choose a layout manager based on the overall design you want for the container. However, some layouts can be difficult to work with in the designer because they immediately take over placement and resizing of a component as soon as you drop it onto the container. To mitigate this effect while you're developing a UI in the designer, you can use `null` layout or the JBuilder custom layout called `XYLayout`. Both of these leave the components exactly where you place them and let you specify their sizes.

However, there are differences between the two:

- `XYLayout` knows about a component's `preferredSize`, so if you choose the `preferredSize` for the component (-1 value), `XYLayout` adjusts the size of the component to match the look and feel of your system. You can't do this with `null` layout.
- `null` clutters up your code with `setBounds()` calls. `XYLayout` doesn't.

Starting with `null` or `XYLayout` makes prototyping easier in your container. Later, after adding components to the container, you can switch to an appropriate portable layout for your design.

**Important** Be sure to convert all containers from `null` or `XYLayout` before deployment.

Because these layouts use absolute positioning, components do not adjust automatically when you resize the parent containers. These layouts do not adjust to differences in users and systems, and therefore, are not portable layouts. See [“XYLayout” on page 87](#) for more information.

In some designs, you might use nested panels to group components in the main container, using various different layouts for the main container and each of its panels.

Experiment with different layouts to see their effect on the container's components. If you find the layout manager you've chosen doesn't give you the results you want, try a different one or try nesting multiple panels with different layouts to get the desired effect.

For a more detailed discussion of each layout, see the individual topics for each layout in [“Layouts provided with JBuilder” on page 89](#)

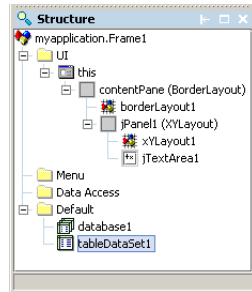
### See also

- [“XYLayout” on page 87](#)
- [“Using nested panels and layouts” on page 120](#)
- [“Laying out components within a container” in the \*Java Language Tutorial\* at <http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html>](#)



## Understanding layout properties

Each container normally has some kind of layout manager attached to its `layout` property. The layout manager has properties that can affect the sizing and location of all components added to the container. These properties can be viewed and edited in the Inspector when the layout manager is selected in the component tree. The layout manager displays as an item in the tree just below the container to which it is attached.



## Understanding layout constraints

For each component you drop into a container, JBuilder may instantiate a `constraints` object or produce a constraint value, which provides additional information about how the layout manager should size and locate this specific component. The type of constraint object or value created depends upon the type of layout manager being used. The Inspector displays the constraints of each component as if they were properties of the component itself, and it allows you to edit them.

## Examples of layout properties and constraints

Below are some examples of layout properties and layout constraints:

- `BorderLayout` has properties called `hgap` (horizontal gap) and `vgap` (vertical gap) that determine the distance between components, while each component in the `BorderLayout` container has a constraint value, called `constraints` in the Inspector, with a possible value of NORTH, SOUTH, EAST, WEST, or CENTER.
- `FlowLayout` and `GridLayout` have properties you can use to modify the alignment of components or the vertical and horizontal gap between them.
- `GridLayout` has properties for specifying the number of rows and columns.
- `GridBagLayout` has no properties itself. However, each component placed into a `GridBagLayout` container has a `GridBagLayoutConstraints` object associated with it that has many properties that control the component's location and appearance, such as
  - The component's height and width
  - Where the component is anchored in its cell
  - How a component fills up its cell
  - How much padding surrounds the component inside its cell

## Using container layouts

JBuilder displays a `layout` property for containers, in the Inspector. You can easily choose a new layout for any container in the designer.

To select a new layout,

- 1 Select the container in the component tree.
- 2 Click the Properties tab in the Inspector and select the `layout` property.
- 3 Click the Down arrow at the end of the `layout` property's value field and choose a layout from the drop-down list.

JBuilder does the following:

- Substitutes the new layout manager in the component tree.
- Changes the source code to add the new layout manager and updates the container's call to `setLayout`.
- Changes the layout of components in the designer.
- Updates the layout constraints for the container's components in the Inspector and in the source code.

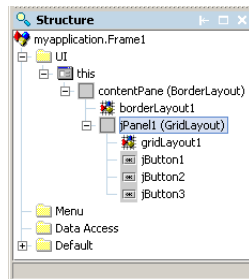
## Modifying layout properties

To modify the properties of a layout from the Inspector,

- 1 Use the component tree to select the layout you want to modify.

JBuilder displays the container's layout directly under each container in the tree.

For example, in the following picture, `gridLayout1` for `jPanel1` is selected.



- 2 Select the Properties page in the Inspector to edit the layout's property values.

For example, in a `GridLayout`, you can change the number of columns or rows in the grid and the horizontal and vertical gap between them.

|                   |             |
|-------------------|-------------|
| <b>name</b>       | gridLayout1 |
| columns           | 0           |
| hgap              | 6           |
| rows              | 1           |
| vgap              | 0           |
| Properties Events |             |

- 3 The designer displays the changes immediately, and JBuilder modifies the source code's `jbInit()` method.

## Modifying component layout constraints

---

When you drop a component into a container, JBuilder creates an appropriate constraint object or value for that container's layout manager. JBuilder automatically inserts this constraint value or object into the `constraint` property of that component in the Inspector. It also adds it to the source code as a parameter of the `add()` method call in the `jbInit()` method.

To edit a component's layout constraints,

- 1 Select the component on the design surface or the component tree.
- 2 Select the `constraints` property in the Inspector.
- 3 Use the drop-down list or property editor to modify the constraints.

## Understanding sizing properties

---

Layout managers use various pieces of information to determine how to position and size components in their containers. AWT components, including Swing components, provide a set of methods that allow layout managers to be intelligent when laying out components. All of these methods are provided so that a component can communicate its desired sizing to the component responsible for sizing it (usually a layout manager).

The methods for this are property getters and represent the following:

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getPreferredSize()</code> | The size a component would choose to be, that is, the ideal size for the component to look best. Depending on the rules of the particular layout manager, the <code>preferredSize</code> may or may not be considered in laying out the container.                                                                                                                                                                                                         |
| <code>getMinimumSize()</code>   | How small the component can be and still be usable. The <code>minimumSize</code> of a component may be limited, for example, by the size of a label. For most of the AWT controls, <code>minimumSize</code> is the same as <code>preferredSize</code> . Layout managers generally respect <code>minimumSize</code> more than they do <code>preferredSize</code> .                                                                                          |
| <code>getMaximumSize()</code>   | The largest useful size for this component. This is so the layout manager won't waste space giving it to a component that can't use it effectively, and instead, giving it to another component that has only its <code>minimumSize</code> . For instance, <code>BorderLayout</code> could limit the center component's size to its maximum size and then either give the space to the edge components or limit the size of the outer window when resized. |
| <code>getAlignmentX()</code>    | How the component would like to be aligned along the x axis, relative to other components.                                                                                                                                                                                                                                                                                                                                                                 |
| <code>getAlignmentY()</code>    | How the component would like to be aligned along the y axis, relative to other components.                                                                                                                                                                                                                                                                                                                                                                 |

To understand how each layout manager uses these pieces of information, study the individual layouts described in [“Layouts provided with JBuilder” on page 89](#).

## Determining the size and location of your UI window at runtime

---

If your UI class is a descendant of `java.awt.Window` (such as a `Frame` or `Dialog`), you can control its size and location at runtime. The size and location is determined by a combination of what the code does when the UI window is created and what the user does to resize or reposition it.

When the UI window is created and various components are added to it, each component added affects the `preferredSize` of the overall window, typically making the `preferredSize` of the window container larger as additional components are added. The exact effect this has on `preferredSize` depends on the layout manager of the outer container, as well as any nested container layouts. For more details about the way that `preferredLayoutSize` is calculated for various layouts, see the sections in this document on each type of layout.

The size of the UI window, as set by your program (before any additional resizing that may be done by the user), is determined by which container method is called last in the code:

- `pack()`
- `setSize()`

The *location* of your UI at runtime will be at the 0,0 position in the top left corner of the screen, unless you override this by setting the `location` property of the container, for example, by calling `setLocation()` before making it visible.

## Sizing a window automatically with `pack()`

---

When you call the `pack()` method on a window, you are asking it to compute its `preferredSize` based upon the components it contains, then size itself to that size. This generally has the effect of making it the smallest it can be while still respecting the `preferredSize` of the components placed within it.

You can call the `pack()` method to automatically set the window to a size that is as small as possible and still have all of the controls and subcontainers on it look good. Note that the `Application.java` file created by the Application wizard calls `pack()` on the frame it creates. This causes the frame to be packed to its `preferredSize` before being made visible.

## Calculating `preferredSize` for containers

---

`preferredSize` is calculated differently for containers with different layouts.

### Portable layouts

Portable layouts, such as `FlowLayout` and `BorderLayout`, calculate their `preferredSize` based on a combination of the layout rules and the `preferredSize` of each component that was added to the container. If any of the components are themselves containers (such as a `Panel`), then the `preferredSize` of that `Panel` is calculated according to its layout and components, the calculation recursing into as many layers of nested containers as necessary.

For more information about `preferredSize` calculation for particular layouts, see the individual layout descriptions.

## XYLayout

For `XYLayout` containers, the `preferredSize` of the container is defined by the values specified in the `width` and `height` properties of the `XYLayout`. For example, if you have the following lines of code in your container initialization,

```
xYLayoutN.setWidth(400);
xYLayoutN.setHeight(300);
```

and if `xYLayoutN` is the layout manager for the container, then its `preferredSize` will be 400 x 300 pixels.

If one of the nested panels in your UI has `XYLayout`, then that panel's `preferredSize` is determined by the layout's `setWidth()` and `setHeight()` calls, and that is the value used for the panel in computing the `preferredSize` of the next outer container.

For example, in the default Application wizard application, the nested panel occupying the center of the frame's `BorderLayout` is itself initially in `XYLayout` and is set to size 400 x 300. This has a significant effect on the overall size of the frame when it is packed, because the nested panel reports its `preferredSize` to be 400x300. The overall frame will be that plus the sizes necessary to satisfy the other components around it in the `BorderLayout` of the frame.

## Explicitly setting the size of a window using `setSize()`

---

If you call `setSize()` on the container (rather than `pack()` or subsequent to calling `pack()`), then the size of the container will be set to a specific size in pixels. This basically has the same effect as the user manually sizing the container: it overrides the effect of `pack()` and `preferredSize` for the container and sets it to some new arbitrary size.

**Important** Although you can certainly set the size of your container to some specific width and height, doing so makes your UI less portable because different screens have different screen resolutions. If you set size explicitly using `setSize()`, you must call `validate()` to get the children laid out properly. (Note that `pack()` calls `validate()`).

## Making the size of your UI portable to various platforms

---

Generally, if you want the UI to be portable, you should use `pack()` and not explicitly `setSize()`, or you should give careful thought to the screen resolutions of various screens and do some reasonable calculation of the size to set.

For example, you may decide that, rather than calling `pack()`, you want to always have the UI show up at 75% of the width and height of the screen. To do this, you could add the following lines of code to your application class, instead of the call to `pack()`:

```
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
frame.setSize(screenSize.width * 3 / 4, screenSize.height * 3 / 4);
```

**Note** Also, to ensure portability, change all `XYLayout` containers to a portable layout after prototyping.

## Positioning a window on the screen

---

If you don't explicitly position your UI on the screen, it appears in the upper left corner of the screen. Often it is nicer to center the UI on the screen. This can be done by obtaining the width and height of the screen, subtracting the width and height of your UI, dividing the difference by two (in order to create equal margins on opposite sides of the UI), and using these half difference figures for the location of the upper left corner of your UI.

An example of this is the code that is generated by the Center Frame On Screen option of the Application wizard. This option creates additional code in the `Application` class which, after creating the frame, positions it in the center of the screen. Take a look at the code generated by this option to see a good example of how to center your UI.

```
//Center the window
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
Dimension frameSize = frame.getSize();
if (frameSize.height > screenSize.height) {
    frameSize.height = screenSize.height;
}
if (frameSize.width > screenSize.width) {
    frameSize.width = screenSize.width;
}
frame.setLocation((screenSize.width - frameSize.width) / 2,
    (screenSize.height - frameSize.height) / 2);
```

## Placing the sizing and positioning method calls in your code

---

The calls to `pack()`, `validate()`, `setSize()`, or `setLocation()` can be made from inside the UI container class, for example, `this.pack()`. They can also be called from the class that creates the container (for example, `frame.pack()` called after invoking the constructor, before the `setVisible()`). The latter is what the Application wizard-generated code does: the calls to `pack()` or `validate()` and `setLocation()` are placed in the `Application` class, after the frame is constructed and the `jbInit()` is therefore finished.

How should you decide where to put your calls for sizing and positioning your UI?

- If you are constructing the UI from various places within your application, and you always want it to come up in the same size and place, you may want to consider putting such calls into the constructor of your UI container class (after the call to `jbInit()`).
- If your application only instantiates the UI from one place, as in the Application wizard-generated application, it is perfectly reasonable to put the sizing and positioning code in the place where the UI is created, in this case the `Application` class.

## Adding custom layout managers

---

JBuilder supports the integration of other layout managers with its designer. To get a custom layout manager to appear in the Inspector's layout property list, create and register a layout assistant for it. There is a sample layout assistant in `samples/OpenToolsAPI/layoutassistant`.

The registration step is a one-line call to the `initOpentool()` static method. Extend `BasicLayoutAssistant` to tell it which layout assistant should handle your layout.

If the custom layout manager uses a constraint class, additional integration can be performed by supplying a class implementing `java.beans.PropertyEditor` for editing the constraint. This property editor would also need to be on the JBuilder classpath.

You would then need to extend `BasicLayoutAssistant` and override two methods:

```
public java.beans.PropertyEditor getPropertyEditor() {
    //return an instance of the constraints property editor
    return new com.mycompany.MyLayoutConstrainedEditor();
}

public String getConstraintsType () {
    // return the fully qualified constraint class name as a string, for example
    return "com.mycompany.MyLayout";
}
```

`BasicLayoutAssistant` is a skeletal implementation of the interface `com.borland.jbuilder.designer.ui.LayoutAssistant`.

Each layout manager must be associated with a class that implements this interface in order for the designer to be able to manipulate the layout. Layouts that do not have a layout assistant can still be used, but you cannot get `BasicLayoutAssistant` assigned to them, and therefore, you cannot move components or convert a container layout to such a layout.

#### See also

- The OpenTools concept documentation
- The `LayoutAssistant` sample, `<jbuilder>/samples/OpenToolApi/LayoutAssistant/LayoutAssistant.jpx`

## Layouts provided with JBuilder

---

JBuilder provides the following layout managers from Java AWT and Swing:

- `BorderLayout`
- `FlowLayout`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`
- `null`
- `XYLayout`, which keeps components you put in a container at their original size and location (x,y coordinates)
- `PaneLayout`, used to control the percentage of the container each of its components occupies.
- `VerticalFlowLayout`, which is very similar to `FlowLayout` except that it arranges the components vertically instead of horizontally
- `BoxLayout2`, a bean wrapper class for Swing's `BoxLayout`, which allows it to be selected as a layout in the Inspector
- `OverlayLayout2`, a bean wrapper class for Swing's `OverlayLayout`, which allows it to be selected as a layout in the Inspector

You can create custom layouts of your own, or experiment with other layouts such as the ones in the `sun.awt` classes or third-party layout managers. Many of these are in the public domain.

**Note** If you want to use a custom layout in the designer, you may have to provide a layout assistant to help the designer use the layout.

Most UI designs will use a combination of layouts by nesting different layout panels within each other. To see how this is done, check the references below.

#### See also

- [“Using nested panels and layouts” on page 120](#)
- “Tutorial: Creating a UI with nested layouts” in online help

## null

---

`null` layout means that no layout manager is assigned to the container. Not specifying a layout means you can put components in a container at specific x,y coordinates relative to the upper left corner of the container.

When designing without a layout manager, specify each component's x,y coordinates in its `constraints` property in the Inspector. Use the tool tip on the design surface: when you grasp a resizable component with the mouse, a tool tip appears which tells you the x,y coordinates in realtime.

Many people find that starting a design without a layout manager simplifies the work of creating an initial design. Once the design takes shape, controlling components with layout managers becomes more meaningful and useful. At that point, switch to an appropriate portable layout for your design.

Be sure to specify a layout manager for the container before deployment, because otherwise components do not adjust well, neither to resizing of the parent container nor to differences in users and systems.

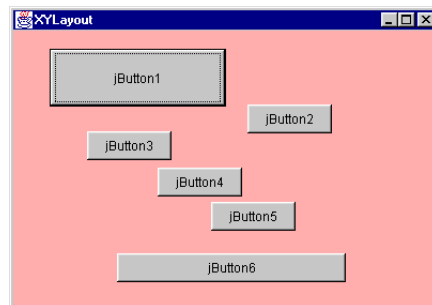
## XYLayout

---

`XYLayout` is a JBuilder custom layout manager. `XYLayout` puts components in a container at specific x,y coordinates relative to the upper left corner of the container. Regardless of the type of display, the container will always retain the relative x,y positions of components. However, when you resize a container with an `XYLayout`, the components **do not** reposition or resize.

**Important** When you change a layout to `XYLayout` in the designer's Inspector, JBuilder adds this import statement to the source code: `com.borland.jbcl.layout.*`. Later, when you complete your UI and change `XYLayout` to a more portable layout before deploying, the import statement is *not* removed. You need to remove it manually if you don't want to import that class.

**Figure 9.1** XYLayout example





`XYLayout` is very convenient for prototyping design work. When you design more complicated user interfaces with multiple, nested panels, `XYLayout` can be used for the initial layout of the panels and components, after which you can choose from one of the standard layouts for the final design.

**Note** To ensure your layout will be nicely laid out on other displays, don't leave any containers in `XYLayout` in your final design.

You can use the visual design tools to specify the container's size and its components' x,y coordinates.

- To specify the size of the `XYLayout` container, select the `XYLayout` object in the component tree and enter the pixel dimension for the `height` and `width` properties in the Inspector. This sets the size of the `XYLayout` container.
- To change the x,y values for a component inside an `XYLayout` container, do one of the following:
  - On the design surface, drag the component to a new size. JBuilder automatically updates the constraint values in the Inspector.
  - Select the component in the component tree, then click the `constraints` property edit field and enter coordinates for that component.

## Aligning components in XYLayout

You can adjust the alignment of a group of selected components in a container that uses `XYLayout`. Alignment does not work for other layouts.

With alignment operations, you can make a set of components the same width, height, left alignment, and so on, so that they look cleanly organized.

To align components,

- 1 Select the components you wish to align. The order of selection affects the alignment.  
See the table below.
- 2 Right-click any selected component on the design surface.
- 3 Select the alignment operation you wish to perform.

### See also

- [Chapter 9, “Using layout managers”](#)

## Alignment options for XYLayout

The following table explains the alignment options available from the context menu:

| Select this   | To do this                                                                                          |
|---------------|-----------------------------------------------------------------------------------------------------|
| Move To First | Move the selected component to the top of the Z-order.                                              |
| Move To Last  | Move the selected component to the bottom of the Z-order.                                           |
| Align Left    | Line up the left edges of the components with the left edge of the first selected component.        |
| Align Center  | Horizontally line up the centers of the components with the center of the first selected component. |
| Align Right   | Line up the right edges of the components with the right edge of the first selected component.      |
| Align Top     | Line up the top edges of the components with the top edge of the first selected component.          |
| Align Middle  | Vertically line up the centers of the components with the middle of the first selected component.   |

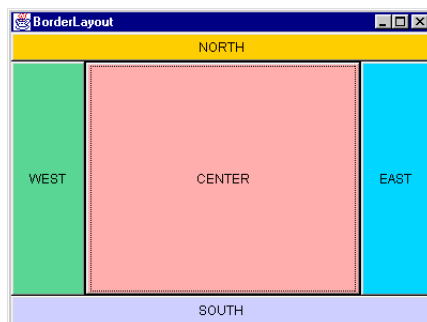
| Select this           | To do this                                                                                       |
|-----------------------|--------------------------------------------------------------------------------------------------|
| Align Bottom          | Line up the bottom edges of the components with the bottom edge of the first selected component. |
| Even Space Horizontal | Evenly space the components horizontally between the first and last selected components.         |
| Even Space Vertical   | Evenly space the components vertically between the first and last selected components.           |
| Same Size Horizontal  | Make the components all the same width as the first selected component.                          |
| Same Size Vertical    | Make the components all the same height as the first selected component.                         |

## BorderLayout

`BorderLayout` arranges a container's components in areas named North, South, East, West, and Center. These are *BorderLayout*'s placement constraints.

- The components in North and South are given their preferred height and are stretched across the full width of the container.
- The components in East and West are given their preferred width and are stretched vertically to fill the space between the North and South areas.
- The component in the Center expands to fill all remaining space.

**Figure 9.2** `BorderLayout`'s placement constraints



JBuilder drops new components into the Center placement of the container by default. Existing components in the container are pushed to the sides as new components are added: first North, then South, then West, then East. Change the placement of a component by selecting the component, selecting the Constraints property in the Inspector, and choosing a different placement from the drop-down list.

There are only five placements available in a `BorderLayout` container. If you need to add more than five components to a container using this layout manager, either nest the components or use a different layout manager. Change a layout manager by selecting the container, selecting the Layout property in the Inspector, and choosing a different layout manager from the drop-down list.

Generally, windows (including frames and dialog boxes) use `BorderLayout` by default.

`BorderLayout` is good for forcing components to one or more edges of a container and for filling up the center of the container with a component. It is also the layout you want to use to cause a single component to completely fill its container.

`BorderLayout` is a useful layout manager for the larger containers in your UI. By nesting a panel inside each area of the `BorderLayout`, then populating each of those panels with other panels of various layouts, you can create rich UI designs.

## Setting constraints

---

For example, to put a toolbar across the top of a `BorderLayout` container, you could create a `FlowLayout` panel of buttons and place it in the North area of the container. You do this by selecting the panel and choosing North for its `constraints` property in the Inspector.

To set the `constraints` property,

- 1 Select the component you want to position, either on the design surface or the component tree.
- 2 Choose the `constraints` property in the Inspector.
- 3 Click the down arrow on the `constraints` property drop-down list and select the area you want the component to occupy.
- 4 Press *Enter*, or click anywhere else in the Inspector, to make the change.

This change is immediately reflected in the code as well as the design.

If you use JBuilder's visual design tools to change the layout of a container from another layout to `BorderLayout`, the components near the edges automatically move to fill the closest edge. A component near the center may be set to Center. If a component moves to an unintended location, you can correct the `constraints` property in the Inspector, or drag the component around on the design surface.

As you drag a component around in a `BorderLayout` container, the design surface displays a rectangle to demonstrate which area of the container the component will snap to if you drop it.

Each of the five areas can contain only one component (or panel of components), so be careful when changing an existing container to `BorderLayout`.

- Make sure the container has no more than five components.
- Use `XYLayout` first to move the components to their approximate intended positions, with only one component near each edge.
- Group multiple components in one area into a panel before converting.

**Note** `BorderLayout` ignores the order in which you add components to the container.

By default, `BorderLayout` puts no gap between the components it manages. However, you can use the Inspector to specify the horizontal or vertical gap in pixels for a layout associated with a container.

To modify the gap surrounding `BorderLayout` components, select the `BorderLayout` object in the component tree (displayed immediately below the container it controls), then modify the pixel value in the Inspector for the `hgap` and `vgap` properties.

### See also

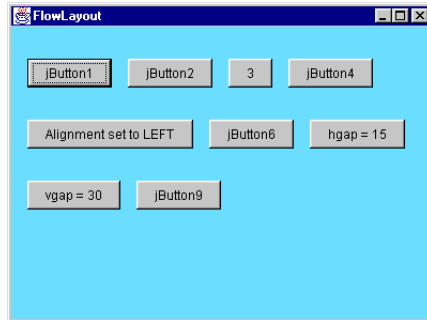
- [“Using nested panels and layouts” on page 120](#)

## FlowLayout

---

`FlowLayout` arranges components in rows from left to right and then top to bottom using each component's natural, `preferredSize`. `FlowLayout` lines up as many components as it can in a row, then moves to a new row. Typically, `FlowLayout` is used to arrange buttons on a panel. In the Java AWT, all panels (including applets) use `FlowLayout` by default.

**Figure 9.3** FlowLayout example



**Note** If you want a panel that arranges the components vertically, rather than horizontally, see [“VerticalFlowLayout” on page 95](#).

You can choose how to arrange the components in the rows of a `FlowLayout` container by specifying an alignment justification of left, right, or center. You can also specify the amount of gap (horizontal and vertical spacing) between components and rows. Use the Inspector to change both the `alignment` and `gap` properties when you're in the designer.

### Alignment

---

The default alignment in a `FlowLayout` is `CENTER`.

- `LEFT` groups the components at the left edge of the container.
- `CENTER` centers the components in the container.
- `RIGHT` groups the components at the right edge of the container.

To change the alignment, select the `FlowLayout` object displayed below the container it controls in the component tree, then specify a value in the Inspector for the `alignment` property.

### Gap

---

The default gap between components in a `FlowLayout` is 5 pixels.

To change the horizontal or vertical gap, select the `FlowLayout` object in the component tree, then modify the pixel value of the `hgap` (horizontal gap) or `vgap` (vertical gap) property in the Inspector.

### Order of components

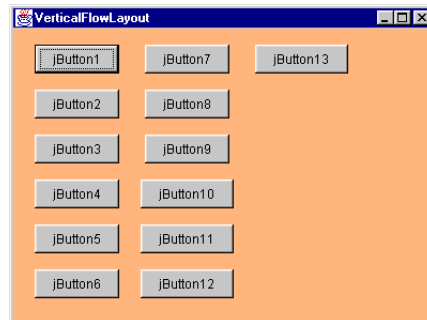
---

To change the order of the components in a `FlowLayout` container, drag the component to the new location, or right-click a component and choose `Move To First` or `Move To Last`.

# VerticalFlowLayout

`VerticalFlowLayout` arranges components in columns from top to bottom, then left to right using each component's natural, preferredSize. `VerticalFlowLayout` lines up as many components as it can in a column, then moves to a new column. Typically, `VerticalFlowLayout` is used to arrange buttons on a panel.

**Figure 9.4** VerticalFlowLayout example



You can choose how to arrange the components in the columns of a `VerticalFlowLayout` container by specifying an alignment justification of top, middle, or bottom. You can also specify the amount of gap (horizontal and vertical spacing) between components and columns. It also has properties that let you specify if the components should fill the width of the column, or if the last component should fill the remaining height of the container. Use the Inspector to change these properties when you're in the designer.

## Alignment

The default alignment in a `VerticalFlowLayout` is TOP.

- TOP groups the components at the top of the container.
- MIDDLE centers the components vertically in the container.
- BOTTOM groups the components so the last component is at the bottom of the container.

To change the alignment, select the `VerticalFlowLayout` object displayed below the container it controls in the component tree, then specify a value in the Inspector for the `alignment` property.

## Gap

The default gap between components in a `VerticalFlowLayout` is 5 pixels.

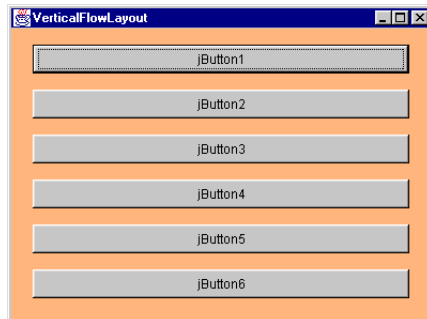
To change the horizontal or vertical gap, select the `VerticalFlowLayout` object in the component tree, then modify the pixel value of the `hgap` (horizontal gap) or `vgap` (vertical gap) property in the Inspector.

## Horizontal fill

---

`horizontalFill` lets you specify a Fill To Edge flag which causes all the components to expand to the container's width.

**Figure 9.5** `horizontalFill` example



**Warning** This causes problems if the main panel has less space than it needs. It also prohibits multi-column output.

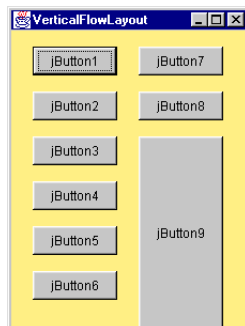
The default value for `horizontalFill` is `True`.

## Vertical fill

---

`verticalFill` lets you specify a Vertical Fill Flag that causes the last component to fill the remaining height of the container.

**Figure 9.6** `verticalFill` example



The default value for `verticalFill` is `False`.

## Order of components

---

To change the order of the components in a `VerticalFlowLayout` container, drag the component to the new location, or right-click a component and choose Move To First or Move To Last.

## BoxLayout2

`BoxLayout2` is Swing's `BoxLayout` wrapped as a Bean so it can be selected as a layout in the Inspector. It combines both `FlowLayout` and `VerticalFlowLayout` functionality into one layout manager.

When you create a `BoxLayout2` container, you specify whether its major axis is the x-axis (left to right placement) or y-axis (top to bottom placement). Components are arranged from left to right (or top to bottom) in the same order as they were added to the container.

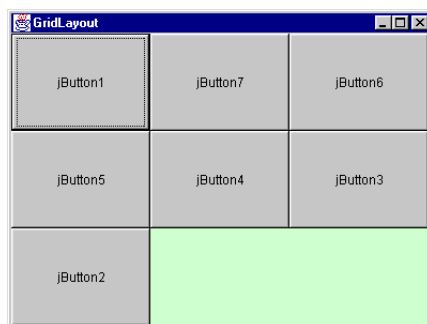
### See also

- `BoxLayout` in the Swing documentation

## GridLayout

`GridLayout` places components in a grid of cells that are in rows and columns. `GridLayout` expands each component to fill the available space within its cell. Each cell is exactly the same size and the grid is uniform. When you resize a `GridLayout` container, `GridLayout` changes the cell size so the cells are as large as possible, given the space available to the container.

**Figure 9.7** `GridLayout` example



Use `GridLayout` if you are designing a container where you want the components to be of equal size, for example, a number pad or a toolbar.

## Columns and rows

You can specify the number of columns and rows in the grid. The basic rule for `GridLayout` is that one of the rows or columns (not both) can be zero. You must have a value in at least one so the `GridLayout` manager can calculate the other.

For example, if you specify four columns and zero rows for a grid that has 15 components, `GridLayout` creates four columns of four rows, with the last row containing three components. Or, if you specify three rows and zero columns, `GridLayout` creates three rows with five full columns.

## Gap

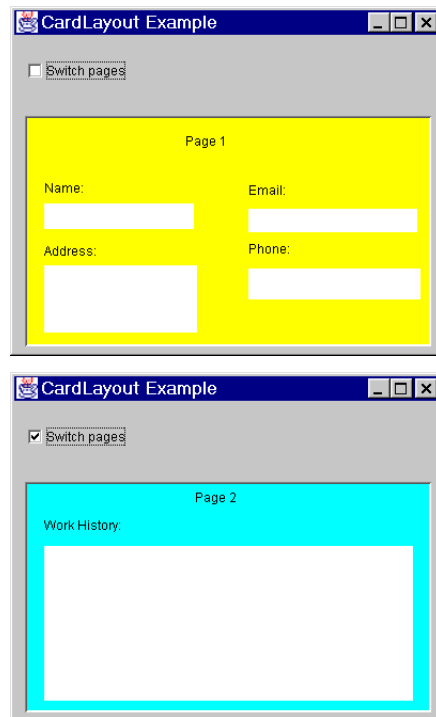
In addition to number of rows and columns, you can specify the number of pixels between the cells using horizontal gap (`hgap`) and vertical gap (`vgap`). The default horizontal and vertical gap is zero.

To change the property values for a `GridLayout` container using the visual design tools, select the `GridLayout` object displayed below the container it controls in the component tree, then edit the values for the `rows`, `cols`, `hgap`, or `vgap` in the Inspector.

## CardLayout

`CardLayout` places components (usually panels) on top of each other in a stack like a deck of cards. You see only one at a time, and you can flip through the panels by using another control to select which panel comes to the top.

**Figure 9.8** CardLayout example



`CardLayout` is a good layout to use when you have an area that contains different components at different times. This gives you a way to manage two or more panels that need to share the same display space.

`CardLayout` is usually associated with a controlling component, such as a checkbox or a list. The state of the controlling component determines which component the `CardLayout` displays. The user makes the choice by selecting something on the UI.

### Creating a CardLayout container

The following example of a `CardLayout` container controlled by a checkbox demonstrates how to create the container in the designer, then hook up a checkbox to switch the panels. This example uses the `JPanel` and `JCheckBox` components from the Swing page of the component palette.

- 1 Create a new project and application with the Application wizard.
- 2 Select the `Frame1.java` file in the project pane, then click the Design tab at the bottom of the AppBrowser to open the UI designer.
- 3 Add a panel (`JPanel1`) to the `contentPane` in the UI designer.
- 4 Set its `layout` property to `XYLayout`.



- 5 Add a panel (`jPanel2`) to the lower half of `JPanel1`.
- 6 Set its layout to `CardLayout`.
- 7 Drop a new panel (`jPanel3`) onto this `CardLayout` panel by clicking `jPanel2` in the component tree. This new panel completely fills up the `CardLayout` panel.
- 8 Change its `background` color property or add UI components to it so it's distinguishable.
- 9 Drop another panel (`jPanel4`) onto `jPanel2` in the component tree. Notice that there are now two panels under `jPanel2` in the component tree.
- 10 Change the background color of `jPanel4` or add components to it.

**Note**

The first component you add to a `CardLayout` panel always fills the panel. To add additional panels to it, click the `CardLayout` panel in the component tree to drop the component, rather than clicking on the design surface.

## Creating the controls

---

Now that you have a stack of two panels in a `CardLayout` container, you need to add a controlling component to your UI, such as a `JList` or `JCheckBox`, so the user can switch the focus between each of the panels.

- 1 Add a `JCheckBox` (`jCheckBox1`) to `jPanel1` that will be used to toggle between the two panels in the `CardLayout` container.
- 2 Select the Events tab in the Inspector for `jCheckBox1` and double-click the `actionPerformed` event to create the event in the source code.
- 3 Add the following code to the `jCheckBox1_actionPerformed(ActionEvent e)` method:

```
if (jCheckBox1.isSelected())
    ((CardLayout)jPanel2.getLayout()).show(jPanel2, "jPanel4");
else
    ((CardLayout)jPanel2.getLayout()).show(jPanel2, "jPanel3");
```

- 4 Compile and run your program.

Click the check box on and off to change the panels in the `CardLayout` container.

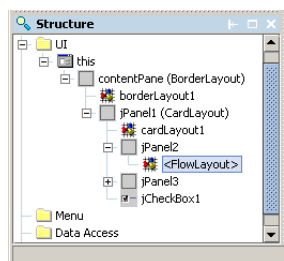
## Specifying the gap

---

Using the Inspector, you can specify the amount of horizontal and vertical gap surrounding a stack of components in a `CardLayout`.

- 1 Select the `CardLayout` object in the component tree.

It's displayed immediately below the container it controls.



- 2 Click `hgap` (horizontal gap) or `vgap` (vertical gap) property in the Inspector.
- 3 Enter the number of pixels you want for the gap.
- 4 Press *Enter* or click anywhere else in the Inspector to register the changes.

## OverlayLayout2

---

`OverlayLayout2` is Swing's `OverlayLayout`, wrapped as a Bean so it can be selected as a layout in the Inspector. It is very much like the `CardLayout` in that it places the components on top of each other.

Unlike `CardLayout`, where only one component at a time is visible, the components can be visible at the same time if you make each component in the container transparent. For example, you could overlay multiple transparent images on top of another in the container to make a composite graphic.

### See also

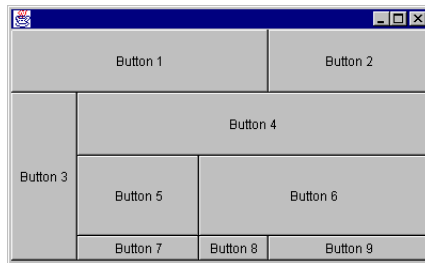
- `OverlayLayout` in the Swing documentation

## GridBagLayout

---

`GridBagLayout` is an extremely flexible and powerful layout that provides more control than `GridLayout` in laying out components in a grid. `GridBagLayout` positions components horizontally and vertically on a dynamic rectangular grid. The components do not have to be the same size, and they can fill up more than one cell.

**Figure 9.9** `GridBagLayout` example



`GridBagLayout` determines the placement of its components based on each component's constraints and minimum size, plus the container's preferred size.

`GridBagLayout` can accommodate either a complex grid or components held in smaller panels nested inside the `GridBagLayout` container. These nested panels can use other layouts and can contain additional panels of components. The nested method has two advantages:

- It gives you more precise control over the placement and size of individual components because you can use more appropriate layouts for specific areas, such as button bars.
- It uses fewer cells, simplifying the `GridBagLayout` and making it much easier to control.

### See also

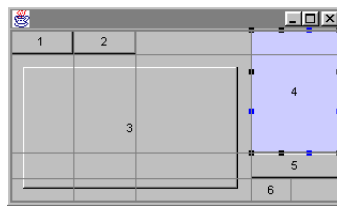
- "Using `GridBagLayout`" in online help for an exploration of `GridBagLayout` and `GridBagConstraints`
- "Tutorial: Creating a `GridBagLayout` in JBuilder" in online help to practice using `GridBagLayout`
- "Tips and techniques" in online help for additional information on using `GridBagLayout` and constraints

## Display area

The definition of a grid cell is the same for `GridBagLayout` as it is for `GridLayout`: a cell is one column wide by one row deep. However, unlike `GridLayout` where all cells are equal in size, `GridBagLayout` cells can be different heights and widths and a component can occupy more than one cell horizontally and vertically.

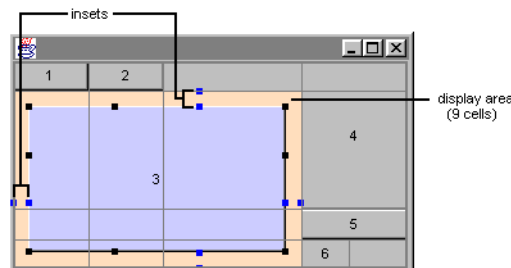
This area occupied by a component is called its *display area*, and it is specified with the component's `GridBagConstraints` `gridwidth` and `gridheight` (number of horizontal and vertical cells in the display area).

For example, in the following `GridBagLayout` container, component “4” spans one cell (or column) horizontally and two cells (rows) vertically. Therefore, its display area consists of two cells.



A component can completely fill up its display area (as with component “4” in the example above), or it can be smaller than its display area.

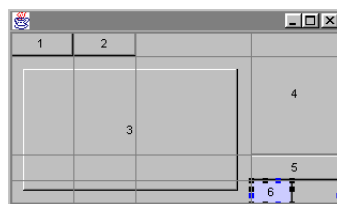
For example, in the following `GridBagLayout` container, the display area for component “3” consists of nine cells, three horizontally and three vertically. However, the component is smaller than the display area because it has insets which create a margin between the edges of the display area and the component.



Even though this component has both horizontal and vertical `fill` constraints, since it also has `insets` on all four sides of the component (represented by the double blue nibs on each side of the display area), these take precedence over the `fill` constraints. The result is that the component only fills the display area up to the `insets`.

If you try to make the component larger than its current display area, `GridBagLayout` increases the size of the cells in the display area to accommodate the new size of the component and leaves space for the `insets`.

A component can also be smaller than its display area when there are no `insets`, as with component “6” in the following example.



Even though the display area is only one cell, there are no constraints that enlarge the component beyond its minimum size. In this case, the width of the display area is determined by the larger components above it in the same column. Component “6” is displayed at its minimum size, and since it is smaller than its display area, it is anchored at the west edge of the display area with an anchor constraint.

As you can see, `GridBagConstraints` play a critical role in `GridBagLayout`. We’ll look at these constraints in detail in the next topic, “About `GridBagConstraints`”.

#### See also

- “How to use `GridBagLayout`” in Sun’s Java tutorial
- “`GridBagLayout`” in the JDK documentation

## About `GridBagConstraints`

---

`GridBagLayout` uses a `GridBagConstraints` object to specify the layout information for each component in a `GridBagLayout` container. Since there is a one-to-one relationship between each component and `GridBagConstraints` object, you need to customize the `GridBagConstraints` object for each of the container’s components.

`GridBagLayout` components have the following constraints:

- anchor
- gridx, gridy
- ipadx, ipady
- gridwidth, gridheight
- fill
- insets
- weightx, weighty

`GridBagConstraints` let you control

- The position of each component, absolute or relative.
- The size of each component, absolute or relative.
- The number of cells each component spans.
- How each cell’s unused display area gets filled.
- How much weight is assigned to each component to control how components utilize extra available space. This controls the components’ behavior when resizing the container.

For a detailed explanation of each of the constraints, including tips for using them and setting them in the designer, see the individual constraint topics below.

#### See also

- “Tutorial: Creating a `GridBagLayout` in JBuilder” in online help for more tips on setting `GridBagConstraints` in the designer

## Setting GridBagConstraints manually in the source code

---

When you use the designer to design a `GridBagLayout` container, JBuilder always creates a new `GridBagConstraints` object for each component you add to the container. `GridBagConstraints` has a constructor that takes all eleven properties of `GridBagConstraints`.

For example,

```
jPanel1.add(gridControl1,
            new GridBagConstraints(0, 0, 1, 1, 1.0, 1.0,
                                   GridBagConstraints.CENTER,
                                   GridBagConstraints.BOTH,
                                   new Insets(35, 73, 0, 0), 0, 0));
jPanel1.add(treeControl1,
            new GridBagConstraints(1, 0, 1, 2, 1.0, 1.0,
                                   GridBagConstraints.CENTER,
                                   GridBagConstraints.BOTH,
                                   new Insets(5, 0, 162, 73), 0, 0));
```

You can modify the parameters of the `GridBagConstraints` constructor directly in the source code, or you can use the `GridBagConstraints` Editor in the designer to change the values.

When you create a `GridBagLayout` container by coding it manually, you really only need to create one `GridBagConstraints` object for each `GridBagLayout` container.

`GridBagLayout` uses the `GridBagConstraints` default values for the component you add to the container, or it reuses the most recently modified value. If you want the component you're adding to the container to have a different value for a particular constraint, then you only need to specify the new constraint value for that component. This new value stays in effect for subsequent components unless, or until, you change it again.

**Note** While this method of coding `GridBagLayout` is the leanest, recycling constraint values from previously added components, it doesn't allow you to edit that container visually in JBuilder's designer.

## Modifying existing GridBagLayout code to work in the designer

---

If you have a `GridBagLayout` container that was previously coded manually by using one `GridBagConstraints` object for the container, you cannot edit that container in the designer without making the following modifications to the code:

- You must create a new JDK 1.3 `GridBagConstraints` object for each component added to the container, which has the large constructor with parameters for each of the eleven constraint values, as shown above.

## Designing GridBagLayout visually in the designer

---

`GridBagLayout` is a complex layout manager that requires some study and practice to understand it, but once it is mastered, it is extremely useful. JBuilder has added some special features to the visual design tools that make `GridBagLayout` much easier to design and control, such as a `GridBagConstraints` Editor, a grid, drag and drop editing, and a context menu on selected components.

There are two approaches you can take to designing `GridBagLayout` in the designer. You can design it from scratch by adding components to a `GridBagLayout` panel, or you can prototype the panel in the designer using another layout first, such as `XYLayout`, then convert it to `GridBagLayout` when you have all the components arranged and sized the way you want.

Whichever method you use, it is recommended that you take advantage of using nested panels to group the components, building them from the inside out. Use these panels to define the major areas of the `GridBagLayout` container. This greatly simplifies your `GridBagLayout` design, giving you fewer cells in the grid and fewer components that need `GridBagConstraints`.

## Converting to GridBagLayout

---

When you prototype your layout in another layout first, such as `XYLayout`, the conversion to `GridBagLayout` is cleaner and easier if you are careful about the alignment of the panels and components as you initially place them, especially left and top alignment. Keep in mind that you are actually designing a grid, so try to place the components inside an imaginary grid, and use nested panels to keep the number of rows and columns as small as possible.

Using `XYLayout` for prototyping gives you the advantage of component alignment functions on the component's context menu.

As the UI designer converts the `XYLayout` container to `GridBagLayout`, it assigns constraint values for the components based on where the components were before you changed the container to `GridBagLayout`. Often, only minor adjustments are necessary, if any.

Converting to `GridBagLayout` assigns `weight` constraints to certain types of components (those which you would normally expect to increase in size as the container is enlarged at runtime, such as text areas, fields, group boxes, or lists). If you need to make adjustments to your design after converting to `GridBagLayout`, you'll find the task much easier if you remove all the `weight` constraints from any components first (set them all to zero).

If even one component has a `weight` constraint value greater than zero, it is hard to predict the sizing behavior in the designer due to the complex interactions between all the components in the container.

You can easily spot a `GridBagLayout` whose components have weights because the components are not clustered together in the center of the container. Instead, the components fill the container to its edges.

**Tip** When you remove all the weights from the components in a `GridBagLayout`, one of two things occur:

- If the container is large enough for the grid, the components cluster together in the center of the container, with any extra space around the edges of the grid.
- If the container is too small for the components, the grid expands beyond the edges of the container and the components that are off the edges of the container are invisible. Just enlarge the size of the container until all the components fit. If the `GridBagLayout` container you are designing is a single panel in the center of the main UI frame, enlarge the size of the frame. You can resize this container to the final size after you have finished setting all the components' constraints.

### See also

- [“GridBagConstraints” on page 107](#) for more details on `weight` constraints

## Adding components to a GridBagLayout container

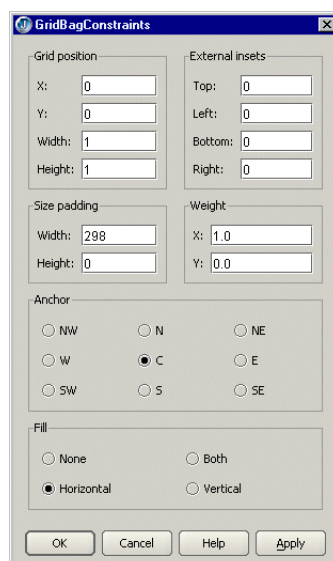
If you want to create your `GridBagLayout` by starting out with a new `GridBagLayout` container and adding all the components to it from scratch, there are certain behaviors you should expect.

- Since the default `weight` constraint for all components is zero, when you add the first component to the container, it locates to the center of the container at its `minimumSize`. You now have a grid with one row and one column.
- The next component you add goes in an adjacent cell, depending on where you click. If you click under the first component, it goes on the next row in that column. If you click to the right of the component, it goes on the same row in the next column. All subsequent components are added the same way, increasing the number of cells by one as you add each one.
- Once you have several components or cells containing components, you can use the mouse to drag the components to new cell locations, or you can change the `gridx` and `gridy` constraints in the `GridBagConstraints` Editor, accessible from the component design surface context menu.
- No matter how many components you add, as long as the grid stays smaller than the container, they all cluster together in the middle of the container. If you need a bigger container, simply enlarge it in the designer.
- If after several rows, your design has been fitting nicely into a certain number of columns, then you suddenly have a row that requires an odd number of components, consider dropping a panel into that row that takes up the entire row, and use a different layout inside that panel to achieve the look you want.

A good example of this is the Sort property editor shown at the end of this section. All of the container's components can fit into two columns except the three buttons at the bottom. If you try to add these buttons individually in the row, `GridBagLayout` does not handle them well. Extra columns are created which affect the placement of the components above it. To simplify the grid and guarantee the buttons would behave the way we expected when the container was resized at runtime, we used a `GridLayout` panel two columns wide to hold the buttons.

## Setting GridBagConstraints in the GridBagConstraints Editor

All the `GridBagConstraints` can be specified in the designer without having to edit the source code. This is possible with the `GridBagLayout` `GridBagConstraints` Editor.



One advantage to using the GridBagConstraints Editor for setting constraints is the ability to change constraints for multiple components at the same time. For example, if you want all the buttons in your `GridBagLayout` container to use the same internal padding, you can hold down the *Shift* key while you select each one, then open the GridBagConstraints Editor and edit the constraint.

To use the GridBagConstraints Editor,

- 1 Select the components within the `GridBagLayout` container you want to modify.  
Select them either in the component tree or on the design surface.
- 2 Do one of the following to open the GridBagConstraints Editor:
  - Select the `constraints` property in the Inspector, then click the ellipsis button.
  - Right-click the component on the design surface and choose Constraints.
  - Select the component in the component tree, press *Shift+F10*, and choose Constraints.
- 3 Set the desired constraints in the property editor, then click OK.

**Note** If you need assistance when using the GridBagConstraints Editor, press the Help button or *F1*.

## Displaying the grid

---

The design surface displays an optional grid that lets you see exactly what is happening with each cell and component in the layout.

- To display this grid, right-click a component in the `GridBagLayout` container and select Show Grid. A check mark is put beside the menu to show that Show Grid is selected.
- To hide the grid temporarily when Show Grid is selected, click a component that is not in the `GridBagLayout` container (including the `GridBagLayout` container itself) and the grid disappears. The grid is only visible when a component inside a `GridBagLayout` container is selected.
- To hide the grid permanently, right-click a component and select Show Grid again to remove the check mark.

## Using the mouse to change constraints

---

The design surface allows you to use the mouse for setting some of the constraints by dragging the whole component or by grabbing various sizing nibs on the component. Directions for setting constraints without the mouse are included in the individual constraint topics below.

## Using the GridBagLayout context menu

---

Right-clicking a `GridBagLayout` component or selecting it and pressing *Shift+F10* displays a context menu that gives you instant access to the GridBagConstraints Editor, and lets you quickly set or remove certain constraints.

| Menu Command   | Action                                                                                                         |
|----------------|----------------------------------------------------------------------------------------------------------------|
| Show Grid      | Displays the <code>GridBagLayout</code> grid in the UI designer.                                               |
| Constraints    | Displays the GridBagConstraints Editor for the selected <code>GridBagLayout</code> component.                  |
| Remove Padding | Sets any size padding values ( <code>ipadx</code> and <code>ipady</code> ) for the selected component to zero. |



| Menu Command      | Action                                                                                                                                                                                                                                             |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Fill Horizontal   | Sets the <code>fill</code> constraint value for the component to <code>HORIZONTAL</code> . The component expands to fill the cell horizontally. If the <code>fill</code> was <code>VERTICAL</code> , it sets the constraint to <code>BOTH</code> . |
| Fill Vertical     | Sets the <code>fill</code> constraint value for the component to <code>VERTICAL</code> . The component expands to fill the cell vertically. If the <code>fill</code> was <code>HORIZONTAL</code> , it sets the constraint to <code>BOTH</code> .   |
| Remove Fill       | Changes the <code>fill</code> constraint value for the component to <code>NONE</code> .                                                                                                                                                            |
| Weight Horizontal | Sets the <code>weightx</code> constraint value for the component to <code>1.0</code> .                                                                                                                                                             |
| Weight Vertical   | Sets the <code>weighty</code> constraint value for the component to <code>1.0</code> .                                                                                                                                                             |
| Remove Weights    | Sets both <code>weightx</code> and <code>weighty</code> constraint values for the component to <code>0.0</code> .                                                                                                                                  |

## GridBagConstraints

The following section lists each of the `GridBagConstraints` separately. It defines each one, explaining its valid and default values, and tells you how to set that constraint visually in the designer.

### See also

- “Tutorial: Creating a GridBagLayout in JBuilder” in online help for more tips on setting `GridBagConstraints` in the designer

### anchor

When the component is smaller than its display area, use the `anchor` constraint to tell the layout manager where to place the component within the area.

The `anchor` constraint only affects the component within its own display area, depending on the `fill` constraint for the component. For example, if the `fill` constraint value for a component is `GridBagConstraints.BOTH` (fill the display area both horizontally and vertically), the `anchor` constraint has no effect because the component takes up the entire available area. For the `anchor` constraint to have an effect, set the `fill` constraint value to `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, or `GridBagConstraints.VERTICAL`.

### Setting the anchor constraint

You can use the mouse to set the `anchor` for a component that is smaller than its cell. Simply click the component and drag it, dragging the component toward the desired location at the edge of its display area, much like you would dock a movable toolbar. For example, to anchor a button to the upper left corner of the cell, click the mouse in the middle of the button and drag it until the upper left corner of the button touches the upper left corner of the cell. This sets the `anchor` constraint value to `NW`.

You can also specify the `anchor` constraint in the `GridBagConstraints` Editor.

- 1 Activate the component’s design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press *Shift+F10*.
- 2 Choose Constraints.
- 3 Select the desired `anchor` constraint value in the Anchor area.
- 4 Click OK.

**fill**

When the component's display area is larger than the component's requested size, use the `fill` constraint to tell the layout manager which parts of the display area should be given to the component.

As with the `anchor` constraint, the `fill` constraint only affects the component within its own display area. `fill` tells the layout manager to expand the component to fill the whole area it has been given.

**Specifying the fill constraint**

The fastest way to specify the `fill` constraint for a component is to use the component's context menu on the design surface.

- 1 Activate the component's design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press *Shift+F10*.
- 2 Do one of the following:
  - Select Fill Horizontal to set the value to HORIZONTAL.
  - Select Fill Vertical to set the value to VERTICAL.
  - Select both Fill Horizontal and Fill Vertical to set the value to BOTH.
  - Select Remove Fill to set the value to NONE.

You can also specify the `fill` constraint in the GridBagConstraints Editor.

- 1 Activate the component's design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press *Shift+F10*.
- 2 Choose Constraints.
- 3 Select the desired `fill` constraint value in the Fill area, then press OK.

**gridwidth, gridheight**

Use these constraints to specify the number of cells in a row (`gridwidth`) or column (`gridheight`) the component uses. This constraint value is stated in cell numbers, not in pixels.

**Specifying gridwidth and gridheight constraints**

You can specify `gridwidth` and `gridheight` constraint values in the GridBagConstraints Editor.

- 1 Activate the component's design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press *Shift+F10*.
- 2 Choose Constraints.
- 3 In the Grid Position area, enter a value for `gridwidth` in the Width field, or a value for `gridheight` in the Height field. Specify the number of cells the component will occupy in the row or column:
  - If you want the value to be `RELATIVE`, enter a -1.
  - If you want the value to be `REMAINDER`, enter a 0.

You can use the mouse to change the `gridwidth` or `gridheight` by sizing the component into adjacent empty cells.

## gridx, gridy

Use these constraints to specify the grid cell location for the upper left corner of the component. For example, `gridx=0` is the first column on the left and `gridy=0` is the first row at the top. Therefore, a component with the constraints `gridx=0` and `gridy=0` is placed in the first cell of the grid (top left).

`GridBagConstraints.RELATIVE` specifies that the component be placed relative to the previous component as follows:

- When used with `gridx`, it specifies that this component be placed immediately to the right of the last component added.
- When used with `gridy`, it specifies that this component be placed immediately below the last component added.

## Specifying the grid cell location

You can use the mouse to specify which cell the upper left corner of the component will occupy. Simply click near the upper left corner of the component and drag it into a new cell. When moving components that take up more than one cell, be sure to click in the upper left cell when you grab the component, or undesired side effects can occur. Sometimes, due to existing values of other constraints for the component, moving the component to a new cell with the mouse may cause changes in other constraint values, for example, the number of cells that the component occupies might change.

To more precisely specify the `gridx` and `gridy` constraint values without accidentally changing other constraints, use the GridBagConstraints Editor.

- 1 Activate the component's design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press *Shift+F10*.
- 2 Choose Constraints.
- 3 In the Grid Position area, enter the column number for `gridx` value in the X field or the row number for `gridy` value in the Y field. If you want the value to be `RELATIVE`, enter a -1.

**Important** When you use the mouse to move a component to an occupied cell, the UI designer ensures that two components never overlap by inserting a new row and column of cells so the components are not on top of each other. When you relocate the component using the GridBagConstraints Editor, the designer does **not** check to make sure the components don't overlap.

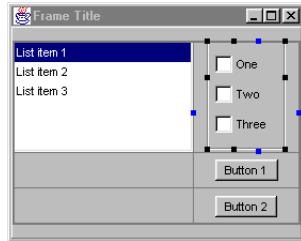
## insets

Use `insets` to specify the minimum amount of external space (padding) in pixels between the component and the edges of its display area. The `inset` says that there must always be the specified gap between the edge of the component and the corresponding edge of the cell. Therefore, `insets` work like brakes on the component to keep it away from the edges of the cell. For example, if you increase the width of a component with left and right `insets` to be wider than its cell, the cell expands to accommodate the component plus its `insets`. Because of this, `fill` and `padding` constraints never steal any space from `insets`.

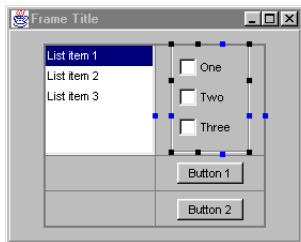
### Setting inset values

The design surface displays blue sizing nibs on a selected `GridBagLayout` component to indicate the location and size of its insets. Grab a blue nib with the mouse and drag it to increase or decrease the size of the inset.

- When an `inset` value is zero, you will only see one blue nib on that side of the cell, as shown below.



- When an `inset` value is greater than zero, the design surface displays a pair of blue nibs for that `inset`, one on the edge of the cell and one on the edge of the display area. The size of the `inset` is the distance (number of pixels) between the two nibs. Grab either nib to change the size of the `inset`.



For more precise control over the `inset` values, use the `GridBagConstraints` Editor to specify the exact number of pixels.

- Right-click the component in the UI designer and choose `Constraints` to display the `GridBagConstraints` Editor.
- In the `External Insets` area, specify the number of pixels for each `inset`: top, left, bottom, or right.

**Note** While negative `inset` values are legal, they can cause components to overlap adjacent components, and are not recommended.

### ipadx, ipady

These constraints specify the internal padding for a component:

- `ipadx` specifies the number of pixels to add to the minimum width of the component.
- `ipady` specifies the number of pixels to add to the minimum height of the component.

Use `ipadx` and `ipady` to specify the amount of space in pixels to add to the minimum size of the component for internal padding. For example, the width of the component will be at least its minimum width plus `ipadx` in pixels. The code only adds it once, splitting it evenly between both sides of the component. Similarly, the height of the component will be at least the minimum height plus `ipady` pixels.

### Example

When added to a component that has a preferred size of 30 pixels wide and 20 pixels high:

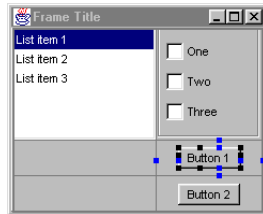
- If `ipadx= 4`, the component is 34 pixels wide.
- If `ipady= 2`, the component is 22 pixels high.

### Setting the size of internal padding constraints

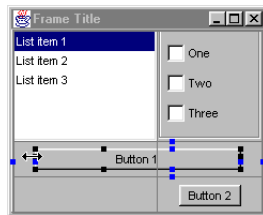
You can specify the size of a component's internal padding by clicking on any of the black sizing nibs at the edges of the component, and dragging with the mouse.

If you drag the sizing nib beyond the edge of the cell into an empty adjacent cell, the component occupies both cells (the `gridwidth` or `gridheight` values increase by one cell).

Before:



After:



For more precise control over the padding values, use the GridBagConstraints Editor to specify the exact number of pixels to use for the value:

- 1 Activate the component's design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press *Shift+F10*.
- 2 Choose Constraints.
- 3 In the Size Padding area, specify the number of pixels for the Width and Height values.

To quickly remove the padding (set it to zero), right-click the component in the UI designer and choose Remove Padding. You can also select multiple components and use the same procedure to remove the padding from all of them at once.

**Note** Negative values make the component smaller than its preferred size and are perfectly valid.

### weightx, weighty

Use the weight constraints to specify how to distribute a `GridBagLayout` container's extra space horizontally (`weightx`) and vertically (`weighty`) when the container is resized. Weights determine what share of the extra space gets allocated to each cell and component when the container is enlarged beyond its default size.

Weight values are of type `double` and are expressed as a ratio. Only positive values are legal. Any ratio format is legal. Mentally assign a total weight to the components in the same row or column, then code a given part of that weight to each component. When you add all of the given weights of all the components together, you should have the total weight you had in mind.

- A row's vertical weight determines the row's height relative to the other rows. This weight equals the largest `weighty` value of the components in the row. The determining factor is height, which is measured on the y axis.

- A column's horizontal weight determines the column's width relative to the other columns. This weight equals the largest `weightx` value of the components in the column. The determining factor is width, which is measured on the x axis.

In theory, only the largest components in a row or column will determine the layout, so you only need one component per row or column that specifies a weight.

### Setting `weightx` and `weighty` constraints

To specify the `weight` constraints for a component in the designer, access the component designer context menu. Either select the component in the tree and press `Shift+F10` or right-click the component, then choose Weight Horizontal (`weightx`), or Weight Vertical (`weighty`). This sets the value to 1.0. To remove the weights (set them to zero), right-click the component and choose Remove Weights. You can do this for multiple components: hold down the `Shift` key when selecting the components, then right-click and choose the appropriate menu item.

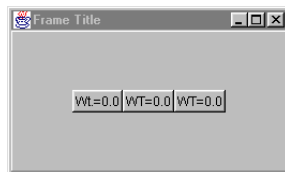
If you want to set the `weight` constraints to be something other than 0.0 or 1.0, you can set the values in the GridBagConstraints Editor:

- 1 Activate the component's design context menu in one of two ways:
  - Right-click the component on the design surface.
  - Select the component in the component tree and press `Shift+F10`.
- 2 Choose Constraints.
- 3 Enter a value between 0.0 and 1.0 for the X (`weightx`) or Y (`weighty`) value in the Weight area, then press OK.

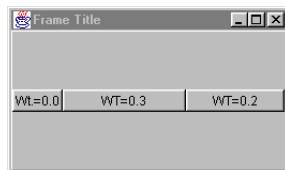
**Important** Because `weight` constraints can make the sizing behavior in the UI designer difficult to predict, setting these constraints should be the last step in designing a `GridBagLayout`.

### Examples of how weight constraints affect components' behavior

- If all the components have `weight` constraints of zero in a single direction, the components clump together in the center of the container for that dimension and won't expand beyond their preferred size. `GridBagLayout` puts any extra space between its grid of cells and the edges of the container.



- If you have three components with `weightx` constraints of 0.0, 0.6, and 0.4 respectively, when the container is enlarged, none of the extra space will go to the first component, 6/10 of it goes to the second component, and 4/10 of it goes to the third.

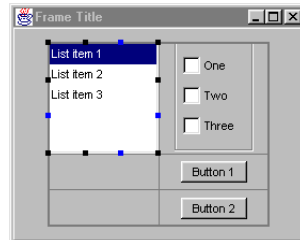


- You need to set both the `weight` and `fill` constraints for a component if you want it to grow. For example, if a component has a `weightx` constraint, but no horizontal `fill` constraint, then the extra space goes to the padding between the left and right edges of the component and the edges of the cell. It enlarges the width of the cell without changing the size of the component. If a component has both `weight` and `fill` constraints, then the extra space is added to the cell, plus the component

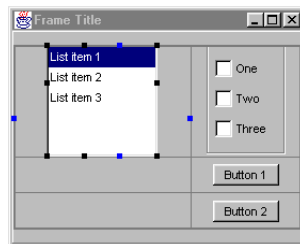
expands to fill the new cell dimension in the direction of the `fill` constraint (horizontal in this case).

The three pictures below demonstrate this.

In the first example, all the components in the `GridBagLayout` panel have a `weight` constraint value of zero. Because of this, the components are clustered in the center of the `GridBagLayout` panel, with all the extra space in the panel distributed between the outside edges of the grid and the panel. The size of the grid is determined by the preferred size of the components, plus any `insets` and padding (`ipadx` or `ipady`).



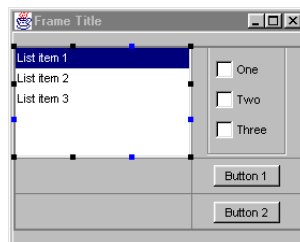
Next, a horizontal `weight` constraint of 1.0 is specified for the `ListControl`. Notice that as soon as one component is assigned any `weight`, the UI design is no longer centered in the panel. Since a horizontal `weight` constraint was used, the `GridBagLayout` manager takes the extra space in the panel that was previously on each side of the grid, and puts it into the cell containing the `ListControl`. Also notice that the `ListControl` did not change size.



#### Tip

If there is more space than you like inside the cells after adding `weight` to the components, decrease the size of the UI frame until the amount of extra space is what you want. To do this, select the `this(BorderLayout)` frame on the design surface or the component tree, then click its black sizing nibs and drag the frame to the desired size.

Finally, if a horizontal `fill` is then added to the `ListControl`, the component expands to fill the new horizontal dimension of the cell.

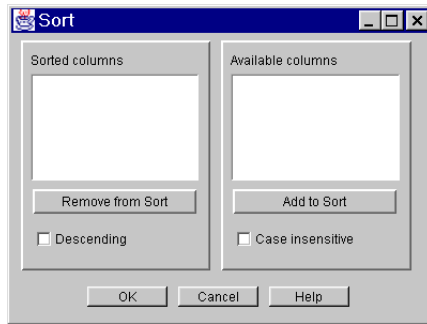


- If one component in a column has a `weightx` value, `GridBagLayout` gives the whole column that weight. Conversely, if one component in a row has a `weighty` value, the whole row is assigned that weight.

#### See also

- “`GridBagConstraints`” in the JDK documentation

## Sample GridBagLayout source code



Notice that, except for the three buttons on the bottom, the rest of the components fit nicely into a grid of two columns. If you try to keep the three buttons in their own individual cells, you would have to add a third column to the design, which means the components above would have to be evenly split across three columns. You could probably have a total of six columns and come up with a workable solution, but the buttons wouldn't stay the same size when the container is resized at runtime.

There are two other ways you could handle this situation.

- Put a `GridLayout` panel that is two columns wide at the bottom of the grid and add the three buttons to it.
- Put a `GridLayout` panel containing the buttons into the outer `BorderLayout` frame. Giving it a constraint of `SOUTH`, and the `BorderLayout` pane a constraint of `CENTER`.

Either way, the resizing behavior of the buttons should be satisfactory.

Here is the relevant source code for this `GridBagLayout` example:

```
package sort;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.borland.jbcl.layout.*;

public class Frame1 extends JFrame {
    JPanel contentPane;
    BorderLayout borderLayout1 = new BorderLayout();
    JPanel jPanel1 = new JPanel();
    JPanel jPanel2 = new JPanel();
    JLabel jLabel1 = new JLabel();
    JList jList1 = new JList();
    JButton jButton1 = new JButton();
    JCheckBox jCheckBox1 = new JCheckBox();
    JButton jButton2 = new JButton();
    JCheckBox jCheckBox2 = new JCheckBox();
    JPanel jPanel3 = new JPanel();
    JList jList2 = new JList();
    JLabel jLabel2 = new JLabel();
    JPanel jPanel4 = new JPanel();
    JButton jButton3 = new JButton();
    JButton jButton4 = new JButton();
    JButton jButton5 = new JButton();
    GridBagLayout gridBagLayout1 = new GridBagLayout();
    GridBagLayout gridBagLayout2 = new GridBagLayout();
    GridBagLayout gridBagLayout3 = new GridBagLayout();
    GridLayout gridLayout1 = new GridLayout();

    //Construct the frame
    public Frame1() {
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```



```

//Component initialization
private void jbInit() throws Exception {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("Sort");
    jPanel1.setLayout(gridBagLayout3);
    jPanel2.setBorder(BorderFactory.createRaisedBevelBorder());
    jPanel2.setLayout(gridBagLayout2);
    jLabel1.setFont(new java.awt.Font("SansSerif", 0, 12));
    jLabel1.setForeground(Color.black);
    jLabel1.setText("Available columns");
    jList1.setBorder(BorderFactory.createLoweredBevelBorder());
    jButton1.setFont(new java.awt.Font("SansSerif", 0, 12));
    jButton1.setBorder(BorderFactory.createRaisedBevelBorder());
    jButton1.setText("Add to Sort");
    jCheckBox1.setText("Case insensitive");
    jCheckBox1.setFont(new java.awt.Font("Dialog", 0, 12));
    jButton2.setText("Remove from Sort");
    jButton2.setBorder(BorderFactory.createRaisedBevelBorder());
    jButton2.setFont(new java.awt.Font("SansSerif", 0, 12));
    jCheckBox2.setFont(new java.awt.Font("Dialog", 0, 12));
    jCheckBox2.setText("Descending");
    jPanel3.setLayout(gridBagLayout1);
    jPanel3.setBorder(BorderFactory.createRaisedBevelBorder());
    jList2.setBorder(BorderFactory.createLoweredBevelBorder());
    jLabel2.setFont(new java.awt.Font("SansSerif", 0, 12));
    jLabel2.setForeground(Color.black);
    jLabel2.setText("Sorted columns");
    jButton3.setText("Help");
    jButton4.setText("OK");
    jButton5.setText("Cancel");
    jPanel4.setLayout(gridLayout1);
    gridLayout1.setHgap(10);
    gridLayout1.setVgap(10);
    contentPane.add(jPanel1, BorderLayout.CENTER);
    jPanel1.add(jPanel2, new GridBagConstraints(1, 0, 1, 1, 1.0, 1.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(6, 10, 0, 19), 0, 2));
    jPanel2.add(jList1, new GridBagConstraints(0, 1, 1, 1, 1.0, 1.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.BOTH,
        new Insets(0, 7, 0, 9), 160, 106));
    jPanel2.add(jButton1, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.NONE, new Insets(8, 7, 0, 9), 90, 2));
    jPanel2.add(jCheckBox1, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(11, 13, 15, 15), 31, 0));
    jPanel2.add(jLabel1, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
        ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(4, 7, 0, 15), 54, 8));
    jPanel1.add(jPanel3, new GridBagConstraints(0, 0, 1, 1, 1.0, 1.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(6, 9, 0, 0), 0, 2));
    jPanel3.add(jList2, new GridBagConstraints(0, 1, 1, 1, 1.0, 1.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.BOTH,
        new Insets(0, 7, 0, 9), 160, 106));
    jPanel3.add(jButton2, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.NONE, new Insets(8, 7, 0, 9), 50, 2));
    jPanel3.add(jCheckBox2, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(11, 13, 15, 15), 56, 0));
    jPanel3.add(jLabel2, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
        ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(4, 7, 0, 15), 67, 8));
    jPanel1.add(jPanel4, new GridBagConstraints(0, 1, 2, 1, 1.0, 1.0,
        ,GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
        new Insets(15, 71, 13, 75), 106, 0));
    jPanel4.add(jButton4, null);
    jPanel4.add(jButton5, null);
    jPanel4.add(jButton3, null);
}

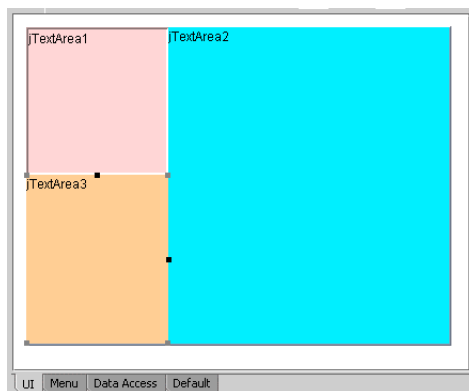
//Overridden so we can exit on System Close
protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
        System.exit(0);
    }
}
}

```

# PaneLayout

`PaneLayout` allows you to specify the size of a component in relation to its sibling components. `PaneLayout` applied to a panel or frame lets you control the percentage of the container the components will have relative to each other, but does not create moveable splitter bars between the panes.

**Figure 9.10** `PaneLayout` example



In a `PaneLayout`, the placement and size of each component is specified relative to the components that have already been added to the container. Each component specifies a `PaneConstraints` object that tells the layout manager from which component to take space, and how much of its existing space to take. Each component's `PaneConstraints` object is applied to the container as it existed at the time the component was added to the container. The order in which you add the components to the container is very important.

## PaneConstraints variables

A `PaneConstraints` component has a constraint that consists of four variables: `String` name, `String` splitComponentName, `String` position, and `float` proportion.

|                                                        |                                                                                                                       |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>String</code> name                               | The name for this component (must be unique for all components in the container — as in <code>CardLayout</code> ).    |
| <code>String</code><br><code>splitComponentName</code> | The name of the component from which space will be taken to make room for this component.                             |
| <code>String</code> position                           | The edge of the <code>splitComponentName</code> to which this component will be anchored.                             |
| These values place the component:                      |                                                                                                                       |
| <code>PaneConstraints.TOP</code>                       | Above <code>splitComponentName</code> .                                                                               |
| <code>PaneConstraints.BOTTOM</code>                    | Below <code>splitComponentName</code> .                                                                               |
| <code>PaneConstraints.RIGHT</code>                     | To the right of <code>splitComponentName</code> .                                                                     |
| <code>PaneConstraints.LEFT</code>                      | To the left of <code>splitComponentName</code> .                                                                      |
| <code>PaneConstraints.ROOT</code>                      | This component is the first component added.                                                                          |
| <code>float</code> proportion                          | The proportion of <code>splitComponentName</code> that will be allocated to this component. A number between 0 and 1. |

## How components are added to PaneLayout

---

`PaneLayout` adds components to the container in the following manner:

- The first component takes all the area of the container. The only important variable in its `PaneConstraint` is its `name`, so the other components have a value to specify as their `splitComponentName`.
- The second component has no choice in specifying its `splitComponentName`. It must use the `name` of the first component.
- The `splitComponentName` of subsequent components may be the `name` of any component that has already been added to the container.

## Creating a PaneLayout container in the designer

---

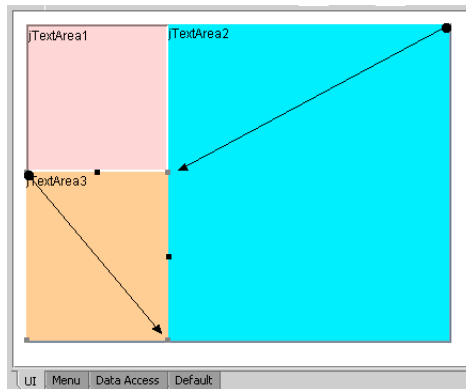
To create and populate a `PaneLayout` container,

- 1 Add a container to your UI in the designer.
- 2 Change the container's `layout` property to `PaneLayout`.  
This allows you to access the `PaneLayout` properties in the Inspector and change the width of the splitter bars.
- 3 Drop a component into the `PaneLayout` container.  
This component completely fills the container, until you add another component to split this one.
- 4 Select another component. Size it if necessary.  
To size it using the mouse,
  - a Place the cursor at the edge or corner which will anchor part of the component.
  - b Hold down the mouse button and drag the cursor across to the corner or edge of the original component, to anchor it in two dimensions. One of the anchors defines where you want to split the area between the components.
 To size the component using the keyboard,
  - c Navigate to the `constraints` property in the Inspector.
  - d Press the *Spacebar* to activate the Inspector cell for the `constraints` property.
  - e Press *Ctrl+Tab* to move focus to the ellipsis button (...) in the Inspector cell.
  - f Press the *Spacebar* to activate the `constraints` property editor.
  - g Within the `PaneConstraints` property editor, use the keyboard to specify the component's position within the pane as well as the proportion of space that the component takes up.
- 5 To add a third component to the `PaneLayout`, draw it similarly to define its relative position to the other components.
- 6 Use the same method to add subsequent components.

For example,

- If you want the panes to be vertical between a right and left half of the panel, drag the mouse starting from the top left corner of the panel to the middle of the bottom edge.
- If you want the panes to be horizontal between an upper and lower half, drag the mouse starting from the top left corner to the middle of the right edge.
- If you want to split the left half of the container, begin drawing the third component starting from the middle of the left edge of the panel to bottom left corner of the second component.

In the image below, the arrows indicate the cursor paths used to obtain the illustrated component sizes. To make the edges of the components more visible in this image, we added a beveled border to each component.



**Important** If the first component you added to a `PaneLayout` container was itself a container, the UI designer assumes you are trying to add the second component to the outer container instead of to the `PaneLayout` container. To specify to the UI designer that you want to add components to containers other than those at the top of the Z-order, select the target container, then hold down the *Ctrl* key while clicking or dragging the component on the design surface.

## Modifying the component location and size in the Inspector

You can use the Inspector to modify which `splitComponent` edge a component should be anchored to and the proportion of the `splitComponent` this component should occupy.

To do this,

- 1 Select the component.
- 2 In the Inspector, select the `constraints` property, then click the ellipsis button (...) to open the Constraints property editor.
- 3 Select one of the following positions: Top, Bottom, Left, Right or Root.  
These values are relative to the component named in the Splits field in the property editor.
- 4 Specify the proportion of the split component this component should occupy.
- 5 Press OK.

**Note** You can also resize the component by selecting it and dragging on the nibs. Moving a component is also allowed, however this will change the add order of the components.

# Prototyping your UI

---

Before you start creating your UI, you may want to sketch your UI design on paper to get an idea of the overall strategy you'll use for placing various panels and components and for assigning layouts. You can also prototype your UI directly in the designer. JBuilder provides a `null` layout assistant and `XYLayout`, which make the initial design work easier.

## Use null or XYLayout for prototyping

---

When you initially add a new panel of any type to the designer, you'll notice that the `layout` property in the Inspector says `<default layout>`. This means the designer will automatically use the default layout for that container. However, you should immediately change the `layout` property to the layout manager you want to use so it is visible in the component tree and its constraints can be modified in the Inspector. You cannot edit layout properties for `<default layout>`.

To control the layout of your components during prototyping, switch each container to `XYLayout` or `null` layout as soon as you drop it into your design. These layouts use pixel coordinates to position the components. This means the components you add to an container will stay at the location you drop them and at the size you specify with the mouse.

### See also

- [“Layouts provided with JBuilder” on page 89](#) for more information on layout constraints

## Design the big regions first

---

We recommend that you start designing the big regions of your UI first, then work down into finer details within those regions as you go, using `XYLayout` or `null` layout exclusively. Once the design is right, work systematically from the inner regions outward, converting the panels to more portable layouts such as `FlowLayout`, `BorderLayout`, or `GridLayout`, making minor adjustments if necessary.

Usually, you place a container in your design first, then add components to it. However, you can also draw a new container around existing components, although these components won't automatically nest into the new panel. After drawing the container, you must explicitly move each component in the container. You may even need to move it out of the container, then back in. Watch the component tree to see when it nests properly. Each component inside a container is indented in the component tree under its container. If the component is at the same level of indentation with a panel, it is not inside it yet.

## Save before experimenting

---

You should expect that when you start designing in JBuilder, you will inevitably do things by trial and error, especially once you start changing the layouts to something other than `XYLayout` or `null` layout. Be sure to save your file before experimenting with a layout change. Then if it doesn't work, you can go back. If you have version control, use it well, especially when using unfamiliar layout managers.

Even when you plan your UI first, you may discover that a particular layout you planned to use just doesn't work as you expected. This might mean reworking the design and using a different configuration of containers, components, and layouts. For this reason, you might want to copy the container file (for example `Frame1.java`) to a different name and location at critical times during the design process, so you won't have to start over.

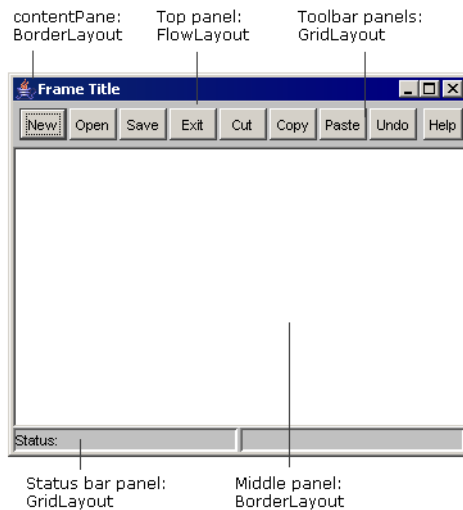
One thing that will speed up your UI design work in the future is to create separate JavaBean components, such as toolbars, status bars, check box groups, or dialog boxes, that you can add to the component palette and reuse with little or no modifications.

## Using nested panels and layouts

---

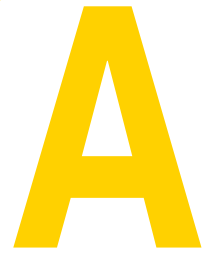
Most UI designs in Java use more than one type of layout to get the desired results by nesting multiple panels with different layouts in the main container. You can also nest panels within other panels to gain more control over the placement of components. By creating a composite design and by using the appropriate layout manager for each panel, you can group and arrange components in a way that is both functional and portable.

For example, the following UI example demonstrates the use of nested panels with different layouts: BorderLayout, FlowLayout, and GridLayout. The entire UI is contained in a `ContentPane` using `BorderLayout`



### See also

- “Tutorial: Creating a UI with nested layouts” in online help to work through the tutorial that builds this UI
- [Chapter 6, “Creating user interfaces”](#) for more about using the designer



## Migrating files from other Java IDEs

JBuilder allows you to migrate your files and applications developed in other Java IDEs into JBuilder. In some cases, you need to modify your code so the file can be visually designed using JBuilder's visual design tools. Java files must meet certain requirements to be visually designable.

To see an example of a visually designable file, create a JBuilder project (File/New Project) and use the Application wizard (File/New) to create a new application.

### See also

- [“Requirements for a class to be visually designable” on page 2](#)

## VisualAge

---

JBuilder is tested against VisualAge version 4.5.1. No modifications are necessary for these files. JBuilder's visual design tools can recognize these files as long as they meet the requirements of a visually designable file. Use the Project For Existing Code wizard to create a new JBuilder project that imports your existing source tree. The Project For Existing Code wizard is a feature of JBuilder Developer and Enterprise.

### See also

- “Creating a project from existing files” in *Building Applications with JBuilder*

## Forte

---

Java files created in Forte need to be modified in the following manner:

- 1 Create a `jbInit()` method.
- 2 Put all of your UI initialization code in the `jbInit()` method.

This includes code that adds components to the container and sets event handling, but does not include code that executes event handling.

- 3 Put all component declarations outside of the `jbInit()` method at the class level.

## VisualCafé

---

The Import VisualCafé Project wizard automates the process of bringing a project created in VisualCafé into JBuilder. This wizard is available from the object gallery's Project page.

To import a project from VisualCafé,

- 1 Choose File|New.

The object gallery appears.

- 2 Select the Project page.
- 3 Select the Import VisualCafé Project icon.
- 4 Either double-click the icon, click OK, or press *Enter* to open the wizard.

### See also

- "Import VisualCafé Project wizard" topic in online help. Either press the Help button in the wizard or choose Help|Help Topics, select the Find page, and type in `visualcafe`.



# Index

## A

- accelerator key, adding to menus 67
- accessibility
  - adding UI components 36
  - designer, keyboard commands 33
  - modifying components in the Inspector 118
  - property editors in the Inspector 117
  - visual designer shortcut keys 33
  - visual designer, navigating in 33
- actionPerformed() menu event 70
- adapter types 49
- adding components 36
  - to component palette 74
  - to database 59
  - to GridBagLayout 105
  - to nested containers 54
  - to UIs 56
- adding events to JavaBeans 17, 19, 20
- adding menus 65, 66
- adjusting frame runtime dimensions 86
- aligning components
  - FlowLayout 94
  - in columns 95
  - in rows 94
  - VerticalFlowLayout 95
  - XYLayout 91
- alignmentX property 85
- alignmentY property 85
- anchor constraints 107
  - setting 107
- anonymous inner class adapters 47, 48
  - choosing 49
- Application wizard, generated UI files 3
- AWT components, compared 4

## B

- Bean Chooser, adding components 30
- BeanInfo classes 15
  - creating 15
  - generating automatically 16
  - modifying 16
- BeanInfo data 15
  - modifying 16
- BeanInfo designer 16
- BeanInsight 25
- BeansExpress 9
  - changing BeanInfo classes 16
  - changing properties 13
  - creating JavaBeans 10
  - removing properties 14
  - setting properties 11, 14
- BorderLayout 92
  - constraints, setting 93
- Borland
  - contacting 6
  - developer support 6
  - e-mail 8
  - newsgroups 7
  - online resources 7

- reporting bugs 8
- technical support 6
- World Wide Web 7
- bound properties, setting for JavaBeans 14
- bugs, reporting 8
- button events 49

## C

- CardLayout 98
- CardLayout layout manager
  - creating controls 99
  - gaps 99
- CDE/Motif Look & Feel 59
- changing BeanInfo classes 16
- classes, visual design requirements 2
- code generated by events 47, 49
- columnar layouts 95
- component libraries 4
  - component palette 29
- component palette 29
  - adding components 74
  - adding pages 75
  - Bean Chooser button 30
  - button images 75
  - installing JavaBeans 26
  - managing 73
  - removing components 76
  - removing pages 76
  - reorganizing 76
- component tree 31, 35
  - accessibility 36
  - adding components 36
  - changing component name 38
  - icons 39
  - moving components 38
  - opening designers 36
  - viewing class names 39
- components 3
  - aligning 91
  - alignmentX property 85
  - alignmentY property 85
  - arranging in grids 97, 100
  - attaching event handlers 46
  - AWT, Swing, dbSwing 4
  - changing name in tree 38
  - component palette 29
  - containers 4
  - containers for grouping 56
  - cross-platform 9
  - cutting, copying and pasting 37
  - database 56, 59
  - DataExpress 32
  - editing and moving in component tree 31
  - finding in design surface 29
  - grouping 56
  - Inspector 31
  - managing, in component tree 35
  - manipulating in UI designs 37, 55
  - maximumSize 85
  - menu *See* Menu designer

- menus 63
- minimumSize 85
- modifying layout constraints 85
- moving 55
- non-UI 32
- non-UI, data-related 32
- preferredSize 85
- red beans 76
- removing from UI designs 38
- resizing 55
- reusable 9
- serializing 77
- setting properties 42
- setting shared properties 42
- UI *See* UI designer
- viewing class names 39
- Window, Frame, Dialog, Panel 4
- components, adding
  - non-UI to UIs 56
  - to Bean Chooser 30
  - to component palette 73, 74
  - to design 36
  - to nested containers 54
  - to PaneLayout 117
- constrained properties, setting for JavaBeans 14
- constraints
  - BorderLayout 92
  - CardLayout 98
  - FlowLayout 94
  - GridBagLayout 100, 102, 105, 106, 107
  - GridLayout 97
  - modifying layout 83, 85
  - OverlayLayout 100
  - PaneLayout 116
  - setting BorderLayout 93
- containers
  - choosing layouts 81
  - components 56
  - overview 4
  - positioning UI on screen 87
  - preferredSize property 86
  - sizing automatically 86
  - sizing explicitly 87
  - UI screen size 86
  - Window, Frame, Dialog, Panel 4
- controls
  - menu components 32
  - UI components 32
- creating BeanInfo classes 15
- creating custom event sets 20
- creating JavaBeans 9, 10
- creating menus 63, 65
  - adding items 66
  - checkable items 67
  - disabling items 67
  - inserting separators 67
  - keyboard shortcuts 67
  - menu events 70
  - moving items 69
  - moving to submenus 70
  - pop-up menus 71
  - radio button items 68
  - submenus 69
- creating property editors 21
- creating UIs 119
- cross-platform components 9

- customizers 79
- customizing
  - adding pages to component palette 75
  - JavaBeans 15

## D

- Data Access designer *See* DataExpress
- database components, adding to UIs 56, 59
- DataExpress components 32
- dbSwing components, compared 4
- Default designer 32
- default layouts 81
- deleting
  - components from designer 38
  - event handlers 46
- design patterns 9
  - JavaBean 9
- design surface 28
- design tasks 53
- design time look and feel 61
- designer component tree 35
- designer *See* designer, visual
- designer types
  - accessing 36
  - Data Access designer *See* DataExpress
  - Default designer *See* Default designer
  - Menu designer *See* Menu designer
  - UI designer 53
    - See also* UI designer
- designer, visual
  - accessibility 33
  - adding components 36
  - component tree 31
  - cutting, copying and pasting components 37
  - deleting components 38
  - design surface 28
  - keyboard shortcuts 33
  - moving components 38
  - parts of 27
  - red components 76
  - selecting components 54
  - serializing components 77
  - setting properties 41
  - shortcut keys 33
  - status bar 29
  - structure pane in Design view *See* component tree
  - tab order 33
  - types of visual designers *See* designer types
  - undoing/redoing 38
  - using customizers 79
  - viewing component class names 39
  - visual design requirements 2
- designing
  - drag and drop 28
  - GridBagLayout with UI designer 103
  - JavaBean user interfaces 11
  - prototyping UIs 119
- Developer Support 6
- dialog boxes
  - adding to project 57
  - adding to UIs 56, 59
  - creating from snippet 57
  - invoking from menu item 50
  - using one that is not a bean 57
- dialog components, pop-up dialogs 32

- Dialog container 4
- dimensions, runtime UI 86
- disabling menu items 67
- documentation conventions 5
  - platform conventions 6
- drag and drop visual design 28
- drop-down list, no property values 42
- drop-down menus, creating 69

## E

---

- editors, customizers 79
- event adapter classes 49
  - overview 47
- event adapters
  - anonymous inner class 48
  - standard 48
- event handlers 45
  - attaching to components 46
  - button example 49
  - creating 49
  - creating for default event 46
  - deleting 46
  - dialog example 50
  - examples 49
- event sets, creating custom 20
- events 45
  - adapters *See* event adapters
  - adding button events 49
  - adding to JavaBeans 17, 19, 20
  - attaching to menu items 50
  - code generated by 47, 49
  - creating and modifying 39
  - dialog example 70
  - menu item events 70
- examples
  - GridBagLayout source code 103
  - invoking dialog from menu 50
  - menu event handling 70
  - serializing a this object 77
- Expose Superclass BeanInfo option 16
- exposing properties in Inspector 40

## F

---

- files, migrating files from Java development tools 121
- fill constraints 108
  - specifying 108
- FlowLayout 94
  - aligning components 94
  - component order 94
  - gap 94
- fonts 5
  - JBuilder documentation conventions 5
- Forte, migrating files to JBuilder 121
- Frame container 4

## G

---

- gap
  - FlowLayout 94
  - VerticalFlowLayout 95
- generating JavaBeans 10
- getAlignmentX() 85
- getAlignmentY() 85
- getMaximumSize() 85
- getMinimumSize() 85

- getPreferredSize() 85
- grid cell, defined 101
- grid lines, displaying in GridBagLayout 106
- GridBagConstraints 102
  - anchor 107
  - changing 106
  - coding manually 103
  - definition of each constraint 107
  - Editor 105
  - fill 108
  - gridheight 108
  - gridwidth 108
  - gridx 109
  - gridy 109
  - insets 109
  - ipadx 110
  - ipady 110
  - weightx 111
  - weighty 111
- GridBagLayout 100
  - context menu for components 106
  - converting to 104
- GridBagLayout containers
  - adding components 105
  - designing visually 103
  - display area 101
  - displaying grid 106
  - example 114
  - modifying code to be designable 103
- gridheight constraints 101, 108
  - specifying 108
- GridLayout 97
  - columns and rows 97
  - gaps 97
- GridLayout containers 97
- gridwidth constraints 101, 108
  - specifying 108
- gridx constraints 109
  - specifying 109
- gridy constraints 109
  - specifying 109
- grouping components 56

## H

---

- handling events *See* event handlers

## I

---

- icons
  - component palette 75
  - component tree 39
  - specifying JavaBean 16
- image files, component palette 75
- importing from other IDEs 121
- inner class adapters 49
- inset constraints 109
  - setting 110
- Inspector 31, 39
  - behind the scenes 42
  - exposing different levels of properties 41
  - property editors 42
  - saving strings 79
  - setting properties 41
  - setting shared properties 42
  - surfacing property values 40

- installing
  - components on component palette 74
  - JavaBeans on component palette 26
- ipadx constraints 110
  - setting 111
- ipady constraints 110
  - setting 111

## J

---

- Java classes, collections of 9
- Java initialization string 22
- Java Metal Look & Feel 59
- JavaBeans 10
  - adding events 17, 19, 20
  - advantages 9
  - Bean Chooser 30
  - changing properties for 13
  - coding visually *See* designer, visual
  - component palette 29
  - containers 4
  - creating 9, 10
  - customizing 15
  - cutting, copying, pasting 37
  - defined 9
  - designing user interface 11
  - displaying property settings 16
  - Inspector 31
  - installing 26
  - removing properties for 14
  - serializing 25
  - setting properties for 11, 14
  - validity 25
  - wizard 10
- JavaBeans *See* components
- JBuilder
  - newsgroups 7
  - reporting bugs 8
- JBuilder, component libraries 4
- JFileChooser dialog box 50

## K

---

- keyboard shortcuts designer 33
- keystrokes, visual designer shortcut keys 33

## L

---

- layout constraints 83
  - changing in GridBagLayout 106
  - examples 83
  - grids 102
  - setting with GridBagConstraints Editor 105
- layout managers 81
  - See also* layouts
  - adding custom 88
  - choosing in Inspector 84
  - default layout 81
  - overview 81
  - unassigned 90
- layout properties
  - examples 83
  - modifying 84

- layouts
  - adding custom 88
  - BorderLayout 92
  - BoxLayout2 97
  - CardLayout 98
    - See also* CardLayout
  - choosing in Inspector 81
  - columnar 95
  - combining columns and rows 97
  - constraints examples 83
  - data grids 97
  - default layout 81
  - FlowLayout 94
    - See also* FlowLayout
  - GridBagLayout 100
    - See also* GridBagLayout
  - GridLayout 97
    - See also* GridLayout
  - modifying 83
  - nested 120
  - null 90
  - OverlayLayout 100
  - OverlayLayout2 100
  - PaneLayout 116
    - See also* PaneLayout
  - portable 86
  - properties examples 83
  - prototyping UI 119
  - prototyping with null or XYLayout 119
  - provided with JBuilder 89
  - VerticalFlowLayout 95
  - XYLayout 87, 90
- libraries, components 4
- localizing String property values 79
- look and feel
  - design time 61
  - modifying 59
  - runtime 59
  - runtime vs. design time 59

## M

---

- MacOS Adaptive Look & Feel 59
- maximumSize 85
- Menu designer 32, 63
  - attaching events 70
  - checkable items 67
  - disabling menu items 67
  - inserting or deleting items 66
  - keyboard shortcuts 67
  - moving items 69
  - pop-up menus 32, 71
  - radio button items 68
  - separators 67
  - submenus, creating 69
  - submenus, moving to 70
  - toolbar 65
  - tools 64
- menu events
  - attaching code example 50
  - creating 70
    - See also* creating menus
  - example of 70

- menus 63
  - adding to UIs 56, 59
  - components 32
  - creating 65
    - See also* creating menus
  - designing 63
  - inserting or deleting items 66
  - keyboard shortcuts 67
  - pop-up menus 32
    - See also* Menu designer
  - terminology 64
- Metal Look & Feel 59
- migrating files from other Java development tools 121
- minimumSize 85
- modifying
  - component layout constraints 85
  - layout constraints 83
  - layout properties 83, 84
- moving components in visual designer 38

## N

---

- navigating in the designer 33
- nested layouts 120
- nested menus, creating 69
  - See also* submenus
- New Property Editor dialog box 21
- newsgroups 7
  - Borland and JBuilder 7
  - public 7
  - Usenet 7
- non-UI components, adding to UIs 56, 59
- null layout 90
  - differences between XYLayout 82

## O

---

- object data types, adding to Inspector 42
- OverlayLayout 100

## P

---

- pack() 86
  - sizing containers automatically 86
  - using in code 88
- Palette Properties dialog box 26
- Panel container 4
- PaneLayout 116
  - adding components 117
  - creating in UI designer 117
  - pane location and size 118
  - PaneConstraint variables 116
- panels
  - adding to CardLayout container 98
  - changing in CardLayout 98
  - nesting 120
- pop-up menus 71
- portability, sizing containers 87
- portable layouts 86
- preferredSize 85, 86
- preinstalled components 73

- properties
  - changing 21, 39
  - customizing 15
  - exposing as class level variable 40
  - exposing in Inspector 41
  - hiding 15
  - JavaBean components 11, 14
  - layouts 84
  - modifying layouts 83
  - setting 39, 42
  - setting for multiple components 42
  - setting in Inspector 41
  - surfacing values in Inspector 40
- property editors 15
  - creating 21
  - custom component 24
  - GridBagConstraints 105
  - Java initialization string 22
  - String List 22
  - String Tag List 22
- Property Exposure Level 41
- PropertyChangeSupport class 14
- prototyping UI using null or XYLayout 119

## R

---

- readObject() 25
- red beans 76
- red squares in visual designer 76
- redoing/undoing in visual designer 38
- reporting bugs 8
- resizing components in designer 55
- ResourceBundle, saving String property values 79
- reusable components 9
- row layouts 94
- runtime look and feel 59

## S

---

- scope, property data types 42
- screen size, runtime UI 86
- separators, inserting in menus 67
- serialization, adding support for 25
- serializing components 77
- serializing objects, alternatives 78
- setSize() 86
  - sizing containers explicitly 87
  - using in code 88
- setting container size
  - automatically 86
  - explicitly 87
- setting events 39
- setting properties 39
  - in JavaBeans 11, 14
- shortcut keys
  - adding to menus 67
  - designer 33
- sizing containers
  - for portability 87
  - using pack() 86
  - using setSize() 87

- SplitPanel 117
  - pane location and size 118
- standard event adapters 47, 48
  - choosing 49
- status bars, designer 29
- String property values, saving to ResourceBundle 79
- structure pane, component tree 35
- submenus, creating 69
- superclasses 16
- Support Serialization option 25
- surfacing property values 40
- Swing components, compared 4

## T

---

- tab order: designer, visual 33
- terminology, menu design 64
- testing UIs 61
- third-party components 74
- this object, serializing 78
- toolbars, menu designer 65
- tools, Menu designer 64
- trees, component tree 35
- types, adding object data types to Inspector 42

## U

---

- UI 53
  - See also* designer, visual
- UI components 32
  - See also* UI designer
  - adding to component palette 73
  - grouping 56
  - selecting 54
- UI design 1
  - See also* designer, visual
  - tips 119
- UI designer 32
  - See also* designer, visual
  - adding components 54
  - adding database components 59
  - adding dialog boxes 57
  - adding menus 57
  - adding non-UI components 56
  - grouping components 56
  - moving and resizing components 55
  - pop-up dialogs 32
  - selecting components 54
  - serializing components 77
  - using customizers 79
- UI window runtime size 86
- undoing/redoin in visual designer 38
- Usenet newsgroups 7
- user interfaces 1
  - See also* designer, visual
  - adding components 36, 54
  - adding database components 56, 59
  - adding dialog boxes 56
  - adding menus 56, 65
  - building with wizards 3
  - cutting, copying, and pasting components 37
  - designing 53
    - See also* UI designer
  - designing JavaBean 11

- grouping components 56
- inserting or deleting menu items 66
- look and feel 59
- Menu designer 32
  - See also* designer types
- moving and resizing components 55
- nesting 120
- positioning on screen 87
- prototyping in designer 119
- selecting components 54
- testing at runtime 61
- visual design *See* UI designer

## V

---

- valid JavaBeans, checking for 25
- variables, exposing property at class level 40
- VerticalFlowLayout 95
  - gap 95
  - horizontal fill 96
  - order of components 96
  - vertical fill 96
- visual components 32
  - Menu designer 32
- visual controls *See* Menu designer
- visual controls *See* UI designer
- visual design 1, 27, 35
  - See also* components
  - See also* designer, visual
  - component palette 29
  - containers 4
  - grouping components 56
  - heirarchical view *See* component tree
  - JavaBeans 3
  - Menu designer *See* Menu designer
  - requirements 2
  - UI designer 32
    - See also* UI designer
  - using the design surface *See* designer, visual
  - using the visual designer 27
  - using wizards 3
- visual design *See* designer, visual
- VisualAge, migrating files to JBuilder 121
- VisualCafe, migrating files to JBuilder 122

## W

---

- weight constraints
  - setting weghtx and weighty 112
  - weightx 111
  - weighty 111
- Window container 4
- Windows Look & Feel 59
- windows, positioning on screen 87
- wizards, in visual design 3
- writeObject() 25

## X

---

- XYLayout 87, 90
  - aligning components 91
  - alignment options 91
  - differences between null layout 82
  - prototyping with 119