Designing Applications with JBuilder®

VERSION 8

Borland®
JBuilder®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
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.

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

Printed in the U.S.A.

JBE0080WW21001designui 3E2R1002
0203040506-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter 9
Tutorial: Building a Java text editor 9-1

Tutorials

Chapter

1

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 AppBrowser" in *Introducing JBuilder* to review the layout and terminology of the AppBrowser.

Building Applications with JBuilder contains the following chapters:

- Chapter 2, "Introducing the 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 3, "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 4, "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 5, “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 6, “Designing menus”

  Explains how to create menus using JBuilder’s Menu designer.

- Chapter 7, “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 8, “Using layout managers”

  Explains Java layout managers and describes how to work with each of the layout managers within the UI designer.

- Tutorials:

  - Chapter 9, “Tutorial: Building a Java text editor”

    Create and deploy a real application to load, edit, and save text files.

  - Chapter 10, “Tutorial: Creating a UI with nested layouts”

    Design a user interface with nested panels and layouts.

  - Chapter 11, “GridBagLayout tutorial”

    Learn how to use GridBagLayout. Create a GridBagLayout UI container in 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.

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
`Monospaced type`	Monospaced type represents the following: • text as it appears onscreen • anything you must type, such as "Type `Hello World` in the Title field of the Application wizard." • file names • path names • directory and folder names • commands, such as `SET PATH` • Java code • Java data types, such as `boolean`, `int`, and `long`. • Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events • argument names • field names • Java keywords, such as `void` and `static`
Bold	Bold is used for java tools, bmj (Borland Make for Java), bcj (Borland Compiler for Java), and compiler options. For example: **javac**, **bmj**, **-classpath**.
Italics	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
Keycaps	This typeface indicates a key on your keyboard, such as "Press *Esc* to exit a menu."
[ ]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	Angle brackets are used to indicate variables in directory paths, command options, and code samples. For example, `<filename>` may be used to indicate where you need to supply a file name (including file extension), and <username> typically indicates that you must provide your user name. When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (< >). For example, you would replace `<filename>` with the name of a file, such as `employee.jds`, and omit the angle brackets. **Note:** Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <font color=red> and <ejb-jar>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
Italics, serif	This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, `<url="jdbc:borland:`*jbuilder*`\\samples\\guestbook.jds">`
...	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, `<home>`. • For UNIX and Linux, the home directory can vary. For example, it could be `/user/<username>` or `/home/<username>` • For Windows NT, the home directory is `C:\Winnt\Profiles\<username>` • For Windows 2000, the home directory is `C:\Documents and Settings\<username>`
Screen shots	Screen shots reflect the Metal 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 Technical Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support on 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/
FTP	ftp://ftp.borland.com/
	Technical documents available by anonymous ftp.
Listserv	To subscribe to electronic newsletters, use the online form at: http://info.borland.com/contact/listserv.html
	or, for Borland's international listserver, http://info.borland.com/contact/intlist.html

World Wide Web

Check 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://www.borland.com/techpubs/jbuilder/ (updated documentation and other files)
- http://community.borland.com/ (contains our web-based news magazine for developers)

Borland newsgroups

You can register JBuilder and 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 in the Support Programs page at http://www.borland.com/devsupport/namerica/. Click the "Reporting Defects" link to bring up the Entry Form.

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. 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.

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.

Any file that meets the above requirements can be designed using the component tree and the Inspector. This allows you to visually design 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 private `jbInit()` method.
- This `jbInit()` method is called correctly from the default constructor.

If JBuilder doesn't find this constructor, it adds it. It also adds any imports needed by the component.

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"

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 where it doesn't find it, it uses introspection to discover the bean's characteristic design patterns.

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 7-1

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

- A *window* is a stand-alone, 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 dialog: Prevents input to any other windows in the application until that dialog is dismissed.

  Modeless dialog: Allows information to be entered in both the dialog and the application.

  File dialog — A basic system-independent File Open/Save dialog box that enables access to the file system.

- 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 several libraries of JavaBean components on the component palette for user interface design, including

- Java `AWT`
- Java `Swing`
- JBuilder `dbSwing` in JBuilder Enterprise

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, resulting in sluggish performance. Swing components have the advantage of being lightweight, delivering more sprightly performance. Browser compatibility is less and less of a problem as time goes on. The dbSwing library that JBuilder Enterprise supplies 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 target audience to have.

Chapter

2

Introducing the designer

The designer consists of features that allow you to visually design classes containing default public constructors. The 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 designer, JBuilder opens the type of designer that's appropriate for the active file.

The designer is an OpenTool. Advanced users who want to learn to add a designer type or otherwise customize the designer, choose Help | Help Topics, select the Contents tab, and open OpenTools Documentation. Read the JBuilder Designer/CMT OpenTools Concepts topic under Developing OpenTools. Package com.borland.jbuilder.cmt and Package com.borland.jbuilder.designer APIs are under OpenTools API Reference.

Using the designer

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

- The content pane shows the design surface.

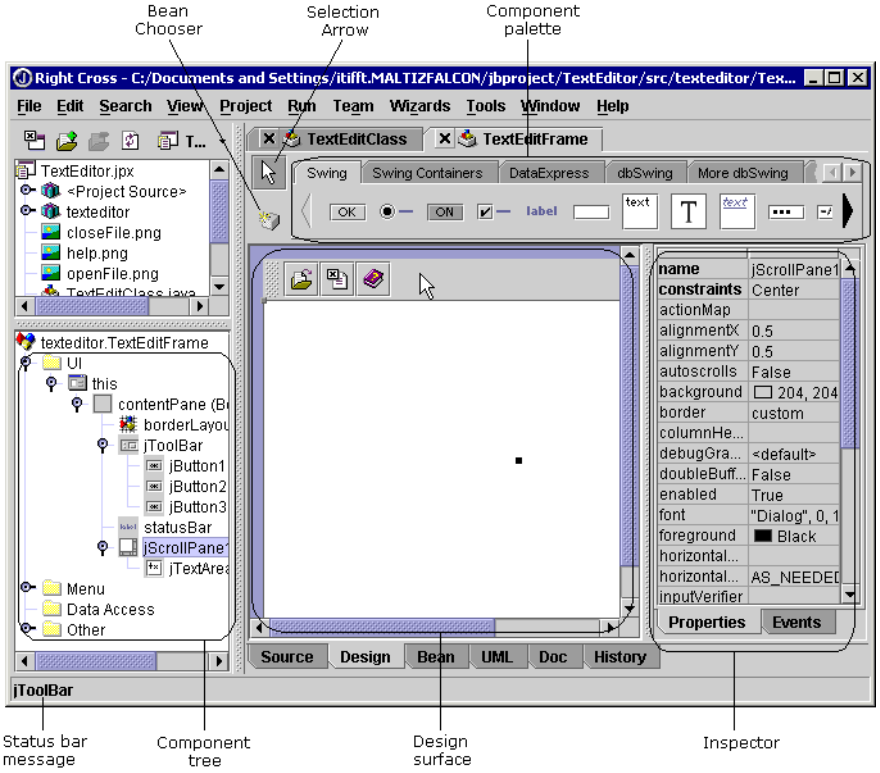
  The content pane also contains:

  - The component palette, displayed at the top of the content pane.
  - The Inspector, displayed on the right margin 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.

Figure 2.1 The designer



In this example,

- The `jScrollPane1` component is selected.

  We can tell because it's highlighted in the component tree, its handle or nib is visible on the design surface, and it's the active component in the Inspector.

- The pointer is on the design surface, hovering over the `jToolBar` 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 `jToolBar` 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 different parts of 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 visible component on the design surface to make it appear in the Inspector, where you can edit its properties.

Select a container on the design surface 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.

Adjust the size of the design surface itself by dragging the left border of the content pane. To hide the AppBrowser's left-hand panes, choose View | Hide All, but keep in mind that this includes the component tree. To bring the left-hand panes back into view, choose View | Show All.

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

See also

- Chapter 8, "Using layout managers"
- "Using the Inspector" on page 3-6
- Chapter 4, "Handling events"

Pinpointing a component

Within a complex design, it can be difficult to tell exactly which of several likely components you're 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 AppBrowser displays the name of the component. This is especially useful if the component you are trying to select is hidden or invisible in the designer, such as a single panel in a `CardLayout` stack. If the panel containing the component is in `XYLayout`, the status bar also displays the x,y coordinates.

See also

- "Selecting components in the UI" on page 5-2 for more on handling components on the design surface.

The component palette

The component palette provides quick access to all available component libraries. Choose the tab containing the types of components you want. Rest your cursor over a component's icon to display a tooltip with the component's name. The component palette is customizable.

Add a component in any of these ways:

- 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.

- 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 3-3
- "Managing the component palette" on page 7-1

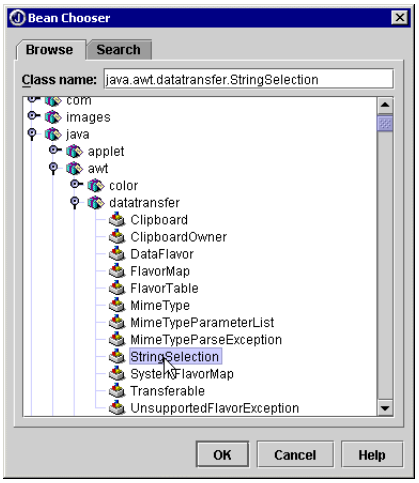
Using the Bean Chooser

The Bean Chooser button is at the left edge of the component palette under 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.

   Notice that a drop-down menu now appears containing the new Java Bean.

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 3-6
- "Using customizers in the designer" on page 7-9

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.

Select a designer type by selecting the appropriate folder in the component tree: Menu, UI, Data Access, or Default. 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 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.

To move a component within the component tree:

1. Select it.
2. Cut it, using either the keyboard shortcut, the Edit menu, or the right-click menu.
3. Select the component that will be immediately above it in its new location.
4. Paste the cut component in.

See also

- "Using the component tree" on page 3-1
- "Using the Inspector" on page 3-6

Designer categories

JBuilder provides individual designers for four broad categories of JavaBeans:

- UI designer
- Menu designer

JBuilder Enterprise

- Data Access designer
- Default designer

Each of these designers provides a set of features that makes designing its particular type of component easier. Change from one designer type to the next by activating components inside the type of designer you want to open. Switch between designer types in one of three ways:

- 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.

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 5, "Creating user interfaces"

Menu designer

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

See also

- Chapter 6, "Designing menus"

Data Access designer

This is a feature of JBuilder Enterprise.

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 5-8
- "DataExpress components" in the *Database Application Developer's Guide*.

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`. 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. When you use the *Tab* or *Ctrl + Tab* keystrokes, focus moves in the following order:

- Project pane
- Component tree
- Component palette
- Design surface (use *Ctrl + Tab* to move out)
- Inspector

Action shortcuts

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

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

Chapter

3

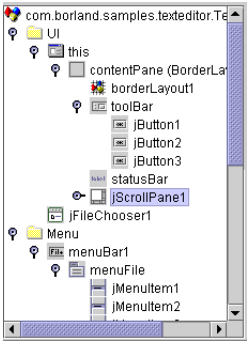
Using the component tree and Inspector

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

Select the folder of the designer 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 become accessible.

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

- Select a parent node in the component tree.
- 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.
- Select or type in the component library and the component you want.
- 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 8, "Using layout managers," for more information on `null`, `XYLayout`, and other layout managers.

Cutting, copying, and pasting components

To cut, copy, or paste components in the designer, select the components in either the design surface or the component tree, then do one of the following:

- Choose the command you want from the Edit menu, or use the appropriate shortcut keys for the function:

Cut	*Ctrl+X*
Copy	*Ctrl+C*
Paste	*Ctrl+V*

- Right-click the selected components, then choose Cut, Copy, or Paste from the context menu.

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 (*Ctrl+Z*) or Edit | Redo (*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 in the component tree, as well as in the Inspector.

To change the name in the 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.

Moving a component

To move a component using the mouse, drag and drop it to its new location. 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 new 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 tooltip.

Understanding component tree icons

The following is an explanation of the icons used to represent the various nodes in the component tree:

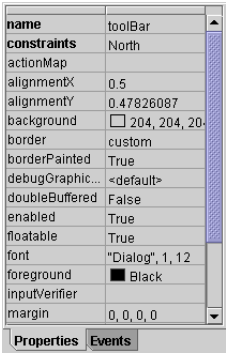
Icon	Explanation
	The current file.
	The layout manager for the parent container.
	The default icon used for a component that doesn't define its own icon.
	Designer type. Can be UI, Menu, Data Access, Default, or custom.

Icons for individual components (panes, buttons, and so on) are generally miniature versions of the icons used in the component palette.

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. View and edit properties on the Properties tab, which displays all supported properties. View and edit events on the Events tab, which displays all supported events.

In the image below, a `toolBar` component is selected and the Properties tab is visible.



The Inspector's left column shows the names of the component properties or events. The right column shows their current values.

Using the Inspector, you can

- Change properties' exposure levels.
- Change property values.
- Set the initial property values for components in the container, and for the container and its layout manager.
- Create event handling code. This creates code to catch events in a container that receives events from a component inside the container.
- Localize strings.

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

See also

- Chapter 4, "Handling events"
- "Using customizers in the designer" on page 7-9 to learn how customizers work with the Inspector.
- Chapter 9, "Tutorial: Building a Java text editor" 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,

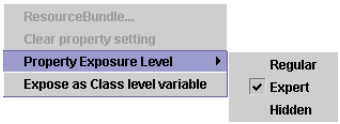
1. Right-click the property in the Inspector.
2. Choose Expose As Class Level Variable from the context menu.

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*.

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 in "Understanding property values".
- "Setting property exposure" below.

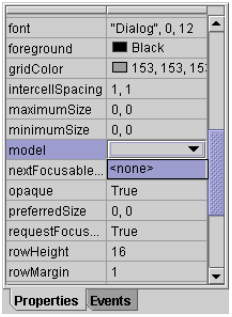
Understanding the Inspector

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 `propertyEditors.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.

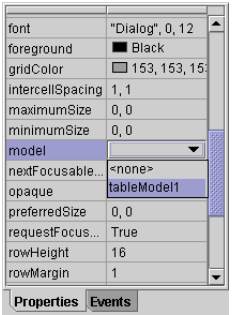
To populate the `model` property drop-down list in the Inspector, add objects of type `TableModel` to your class. For instance, you can add a `TableModel` class from the `javax.swing.table` package from the component palette:

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.

Chapter

4

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

- "Using the Inspector" on page 3-6
- "Requirements for a class to be visually designable" on page 1-6

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, then select the Doc tab.

See also

- "Connecting controls and events" on page 4-3

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.
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 event handling code to match existing event handlers, check the Match Existing Code option.
5. 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, choose the General tab of the object gallery, 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 tab 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`.

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 tab 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 sub-component 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

- Chapter 9, "Tutorial: Building a Java text editor"

Chapter

5

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.
- Setting container layouts and component constraints.

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

See also

- "The Application wizard" in the online help.
- "Adding components" on page 3-3
- "Setting property values" on page 3-8
- "Attaching event-handling code" on page 4-2
- Chapter 8, "Using layout managers"

Selecting components in the UI

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 3-3 for more on how to add single and multiple components.
- "Action shortcuts" on page 2-9

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.

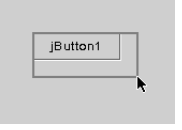
To size a component as you add it,

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.

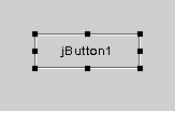


4 Release the mouse button when the outline is the size you want.

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 3-4 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, making sure the mouse pointer falls within the container's boundaries. (The status bar at the bottom of the AppBrowser displays which container your mouse is over.) You can drop a new component from the component palette, or drag an existing component into the new container. As you add components, they appear inside the selected container on the design surface and 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:

Swing containers tab	`JMenuBar` `JPopUpMenu`
AWT tab	`MenuBar` `PopUpMenu`

1 Drop it anywhere on the component tree or on the design surface. Notice that it is placed in the component tree's `Menu` folder.

2 Double-click the menu component in the component tree to open the Menu designer, or right-click it and choose Activate Designer.

3 Add menu items in the Menu designer.

4 Attach events to the menu items by using the Inspector or manually typing the code.

5 Close the Menu designer by double-clicking a UI component in the component tree.

See also

- Chapter 6, "Designing menus"

Dialog boxes

The Dialog wizard is a feature of JBuilder SE and Enterprise.

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 (File | New).

To add an existing dialog,

1 Select one of the dialog components, such as `JFileChooser`, on the component palette. You'll find them on the Swing Containers tab and on the More dbSwing tab in the SE and Enterprise editions.

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 File | New and double-click 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 4, "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

- Chapter 9, "Tutorial: Building a Java text editor," to see examples of using modal dialog box components.

Database components

This is a feature of JBuilder Enterprise.

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 tab 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 Using either technique above, add a component with columns, such as a `TableDataSet` or a `QueryDataSet`.

2 Click the expand icon beside it in the component tree to expand the component.

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 the *Database Application Developer's Guide* 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:

- 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

- "Running Java programs" in *Building Applications with JBuilder*
- "Building Java programs" in *Building Applications with JBuilder*
- "Compiling Java programs" in *Building Applications with JBuilder*
- "Debugging Java programs" in *Building Applications with JBuilder*
- "Deploying Java programs" in *Building Applications with JBuilder*

Chapter

6

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, first select the AppBrowser's Design tab to open the UI designer. Once the designer is open,

1. Expand the Menu folder in the component tree.
2. 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.

1. 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`
2. 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 three ways:

- 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.

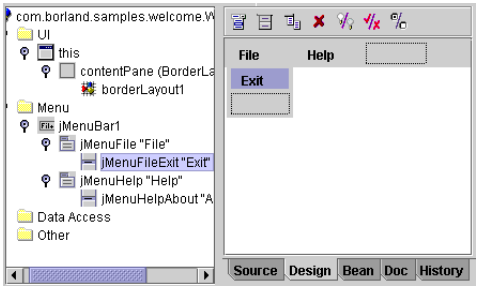
Menu terminology

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 `enabled` property of the selected menu item between `true` and `false` and dims it when it is `false` (disabled). (Applies to Swing menu components; the `enabled` property of AWT menu components must be changed by hand in your source code, since it's not available to the Inspector.)
	Makes the menu item checkable.
	Toggles the menu item between being a `JMenuItem` or a `JRadioButtonMenuItem`.

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 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 a cursor.)

   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 and 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, but not otherwise.

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, set its `enabled` property to `false`. The default state of a menu item is `true`.

To disable a Swing menu item

- 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. This adds a `buttonGroup1` to the Default folder in the tree, and adds the following line to the class variables:

```
ButtonGroup buttonGroup1 = new ButtonGroup();
```

6. Click the Source tab, and, in the constructor's `try` block after `jbInit()`, add each `jRadioButtonMenuItem` to `buttonGroup1` as follows:

```
//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();
      }
    }
```

7 Click the Design tab, and in the Inspector, set the `selected` property for one of the radio button menu items to `true` (whichever one you want selected as the default). The following line of code is added to your source:

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

8 Now, if 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. If 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,

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

   If you are dragging the menu or menu item to another menu, drag it along the menu bar until the cursor points to the target menu. This

opens the target menu, letting you drag and drop the menu or menu item to its new location.

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

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 for which you want to create 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 4-2
- "Connecting controls and events" on page 4-3

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 one or more menu items to the menu.

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

5 Select the panel or other component to whose event you want the pop-up menu attached, so you can see that component in the Inspector. For the example below, `jPanel1` was selected.

6 Click the Event tab in the Inspector.

7 Double-click the event for which you want the pop-up menu to appear. The `MouseClicked` event was selected in the example below.

8 Edit your event-handler skeleton code to 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.

Chapter

7

Advanced topics

The component palette can be customized to include additional component libraries and individual JavaBeans. 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 tab in the component palette displays components in the Swing library.

JBuilder comes with several component libraries. Choose from among these using the different tabs 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*.

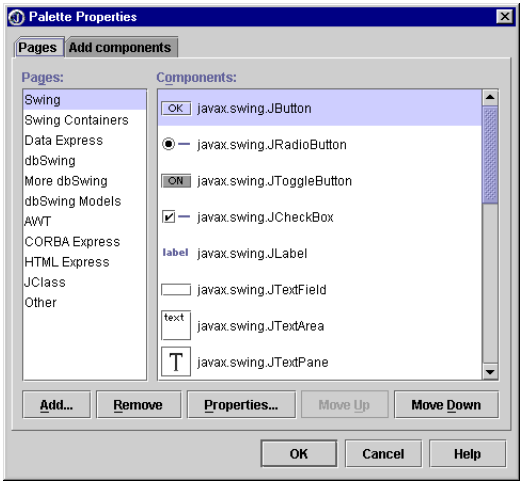
Adding a component to the component palette

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

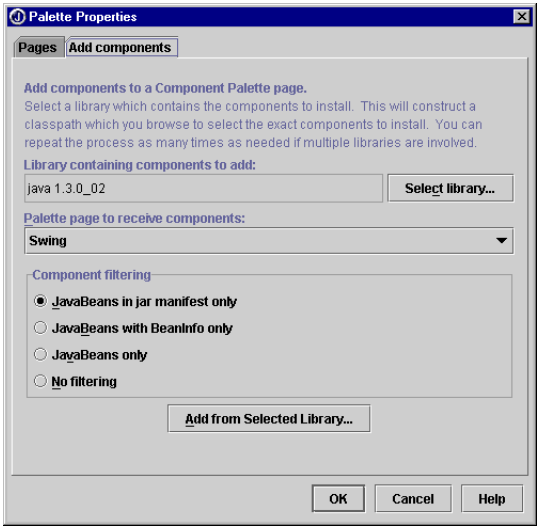
If the component is to be shared between projects, it should be added to a library that is on the `CLASSPATH` in each project where it will be used.

To place the component on the component palette,

1. Choose Tools | Configure Palette, or right-click a palette tab and choose Properties to display the Palette Properties dialog box.
2. Select the Pages tab. In the Pages column, select the palette page on which you want your component to appear, or click the Add button to create a new page.



3 Click the Add Components tab where you select the component(s) you're adding.



4 Press the Select Library button to open the Select A Different Library dialog box. Select an existing library from the list or create a new one by pressing New and using the New Library wizard. Click OK to close the dialog box.

5 Select the palette page to which you want the components added.

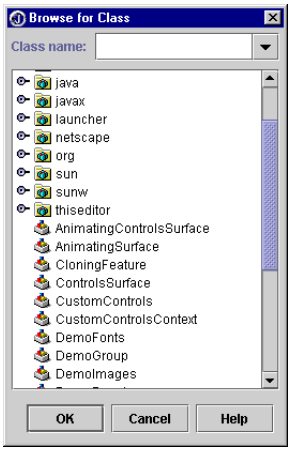
6 Select a filtering option:

- 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.

7 Choose 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.

8 If you selected one of the other options, the Browse For Class dialog box displays. Select the individual classes you want from the Browse For Class dialog box, then click OK. A Results dialog box displays with the classes listed. Click OK.



9 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 Item 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.

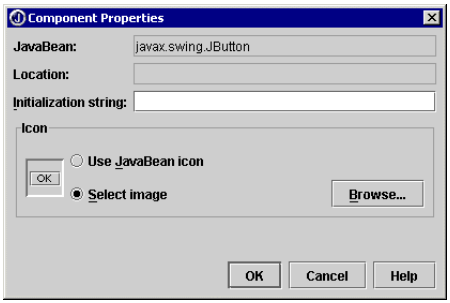
Both of these actions open the Palette Properties dialog box.

2 Select 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.

The Item Properties dialog box appears.



Note An image of the selected component is displayed in the Item 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 select 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 Item Properties dialog box.

7 Click OK in the Palette Properties dialog box when you're finished.

Adding a page 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 Click OK.

6 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.

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/1.3/docs/guide/serialization/`.

Serializing components in JBuilder

JBuilder makes it easy to serialize JavaBeans in the designer. First, modify the bean using the Inspector to give it any settings you want to keep. Then follow these steps:

1. Select and right-click the component in the component tree.
2. 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.
3. 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

The rule for serialization is simple: 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 a RAD tool (like an Inspector) or a customizer to make changes to it.

The following example shows how you can serialize 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

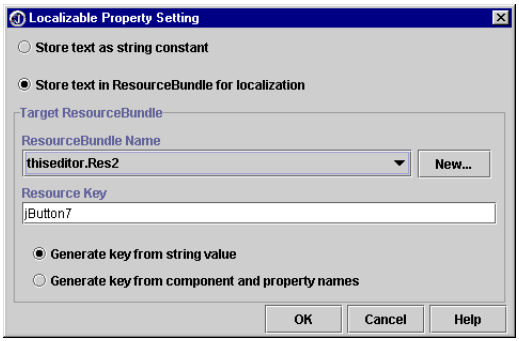
- "Creating JavaBeans with BeansExpress" in *Building Applications with JBuilder*
- "Bean Customization" at `http://java.sun.com/docs/books/tutorial/javabeans/customization/index.html`

Handling resource bundle strings

This is a feature of JBuilder SE 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 displays.



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.

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

1. Select Store Text In ResourceBundle For Localization.
2. Select the existing ResourceBundle file name to use, or click New to create a new one.
3. Select the `ResourceBundle` type and 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) in the online help.

Chapter

8

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 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

XYLayout is a feature of JBuilder SE and Enterprise.

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 "Layout managers: XYLayout" 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.

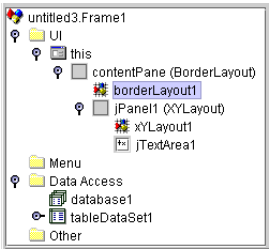
For a more detailed discussion of each layout, see the individual topics for each layout in "Layouts provided with JBuilder" on page 8-11.

See also

- "XYLayout" on page 8-12
- "Using nested panels and layouts" on page 8-51
- "Laying out components within a container" in the *Java Language Tutorial* at `http://java.sun.com/docs/books/tutorial/uiswing/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 constraints 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

Selecting a new layout for a container

JBuilder provides a `layout` property in the Inspector for containers. 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.

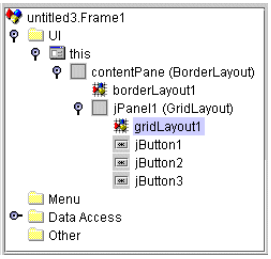
JBuilder does the following:

- 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.

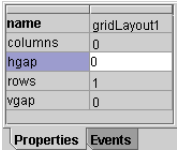
Modifying layout properties

To modify the properties of a layout from the Inspector,

1 Select the layout you want to modify in the component tree. 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 and 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.



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 pull-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:

`getPreferredSize()`	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 `preferredSize` may or may not be considered in laying out the container.
`getMinimumSize()`	How small the component can be and still be usable. The `minimumSize` of a component may be limited, for example, by the size of a label. For most of the AWT controls, `minimumSize` is the same as `preferredSize`. Layout managers generally respect `minimumSize` more than they do `preferredSize`.
`getMaximumSize()`	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 `minimumSize`. For instance, `BorderLayout` 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.
`getAlignmentX()`	How the component would like to be aligned along the x axis, relative to other components.
`getAlignmentY()`	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 8-11.

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

XYLayout is a feature of JBuilder SE and Enterprise.

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 report 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.MyLayoutConstrainteditor();
}
public String getContstraintsType () {
  // 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

- OpenTools Custom Layout Assistants 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

JBuilder SE and Enterprise also provide these custom layouts:

- 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

Each of JBuilder's layout managers is explained in detail later in this section.

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 public domain on the Web. 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 8-51.
- Chapter 10, "Tutorial: Creating a UI with nested layouts"

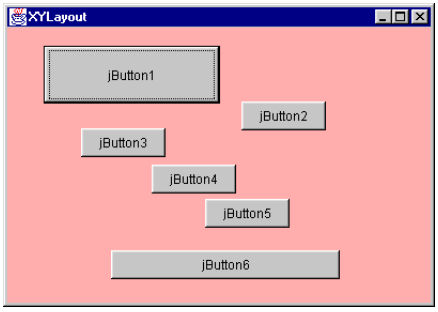
XYLayout

This is a feature of JBuilder SE and Enterprise.

`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 8.1 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, as described in the table below.
2. Right-click a selected component on the design surface and select the alignment operation you wish to perform.

See also

- "Using layout managers" on page 8-1

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.
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.

null

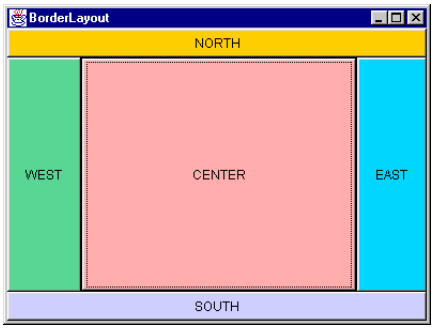
`null` layout means that no layout manager is assigned to the container. Not specifying a layout is very similar to using `XYLayout` in that you can put components in a container at specific x,y coordinates relative to the upper left corner of the container. You must specify each component's x,y coordinates in its `constraints` property. Later, you can 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 when you resize the parent container nor to differences in users and systems.

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 8.2 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.

In the Java AWT, all 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. Select 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 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.

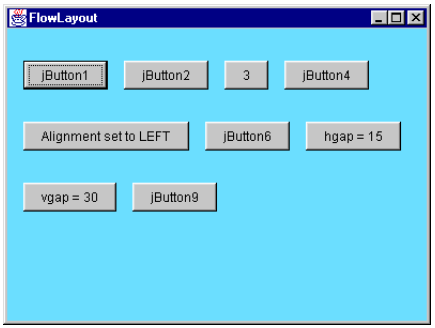
See also

- "Using nested panels and layouts" on page 8-51

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 8.3 FlowLayout example



Note If you want a panel that arranges the components vertically, rather than horizontally, see "VerticalFlowLayout" on page 8-18. VerticalFlowLayout is a feature of JBuilder SE and Enterprise.

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

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.

The default alignment in a FlowLayout is CENTER.

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

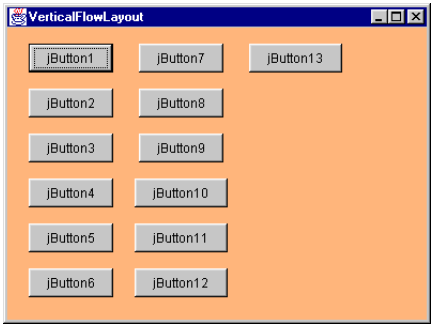
To change the order of the components in a FlowLayout container, drag the component to the new location, or right-click a component and choose Move To First or Move To Last.

VerticalFlowLayout

This is a feature of JBuilder SE and Enterprise.

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 8.4 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

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.

The default alignment in a `VerticalFlowLayout` is TOP.

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

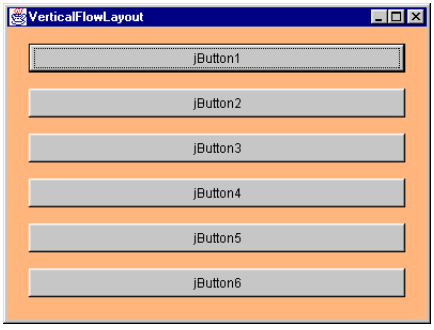
The default gap between components in a `VerticalFlowLayout` is 5 pixels.

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 8.5 horizontalFill example



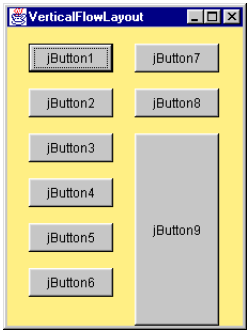
Warning This causes problems if the main panel has less space than it needs. It also prohibits multi-column output.

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 8.6 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

This is a feature of JBuilder SE and Enterprise.

`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.

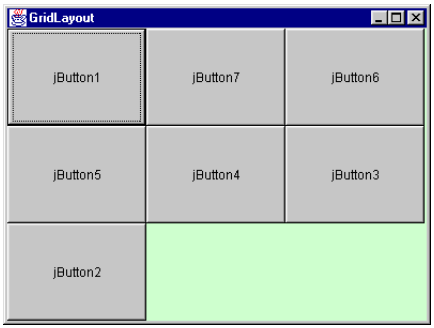
See also

- `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 8.7 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

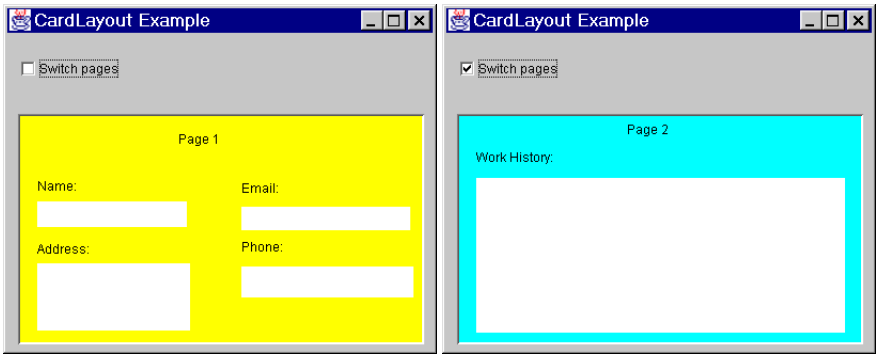
In addition to number of rows and columns, you can specify the number of pixels between the cells using horizontal gap (hgap) and vertical gap (vgap). The default horizontal and vertical gap is zero.

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 8.8 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 check box 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.

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.

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.

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 :

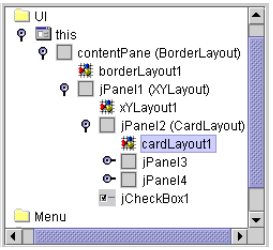
```
if (jCheckBox1.isSelected())
   ((CardLayout)jPanel2.getLayout()).show(jPanel2,"jPanel4");
else
  ((CardLayout)jPanel2.getLayout()).show(jPanel2,"jPanel3");
```

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 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

This is a feature of JBuilder SE and Enterprise.

`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.

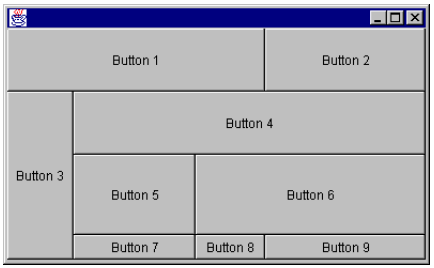
See also

- `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 8.9 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

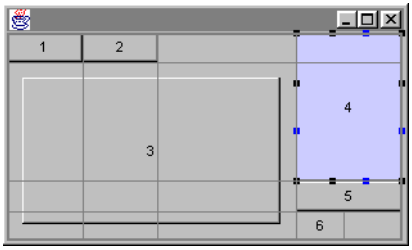
- Chapter 11, “GridBagLayout tutorial”

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.

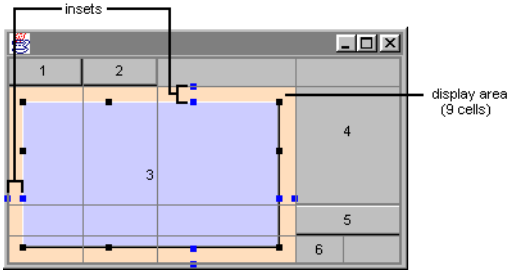
This area occupied by a component is called its `display area`, and it is specified with the component's `GridBagConstraints` gridwidth and gridheight (number of horizontal and vertical cells in the display area).

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.

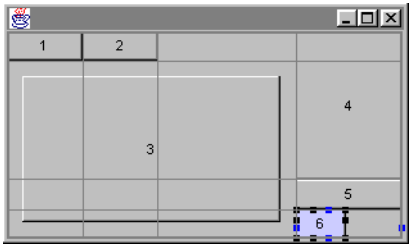
For example, in the following `GridBagLayout` container, the display area for component "3" consists of nine cells, three horizontally and three vertically. However, the component is smaller than the display area because it has insets which create a margin between the edges of the display area and the 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

- Chapter 11, "GridBagLayout tutorial"

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.

XYLayout is a feature of JBuilder SE and Enterprise.

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 8-34 for more details on weight constraints.

Adding components to a GridBagLayout container

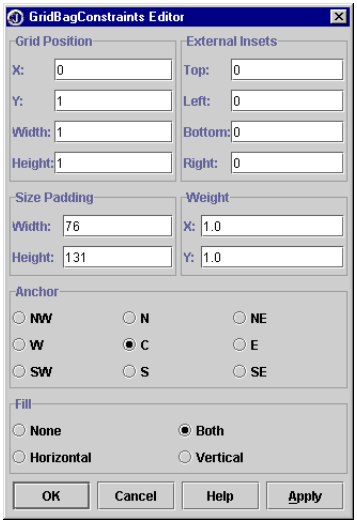
If you want to create your `GridBagLayout` by starting out with a new `GridBagLayout` container and adding all the components to it from scratch, there are certain behaviors you should expect.

- 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 component(s) within the `GridBagLayout` container you want to modify, 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 checkmark.

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 visually 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 `GridBagLayout` grid in the UI designer.
Constraints	Displays the GridBagConstraints Editor for the selected `GridBagLayout` component.
Remove Padding	Sets any size padding values (`ipadx` and `ipady`) for the selected component to zero.
Fill Horizontal	Sets the `fill` constraint value for the component to HORIZONTAL. The component expands to fill the cell horizontally. If the `fill` was VERTICAL, it sets the constraint to BOTH.
Fill Vertical	Sets the `fill` constraint value for the component to VERTICAL. The component expands to fill the cell vertically. If the `fill` was HORIZONTAL, it sets the constraint to BOTH.
Remove Fill	Changes the `fill` constraint value for the component to NONE.
Weight Horizontal	Sets the `weightx` constraint value for the component to 1.0.
Weight Vertical	Sets the `weighty` constraint value for the component to 1.0
Remove Weights	Sets both `weightx` and `weighty` constraint values for the component to 0.0.

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

- Chapter 11, "GridBagLayout tutorial," 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 in the designer

You can use the mouse to set the `anchor` for a component that is smaller than its cell. You 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, then press 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 in the designer

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 in the designer

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 in the designer

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.

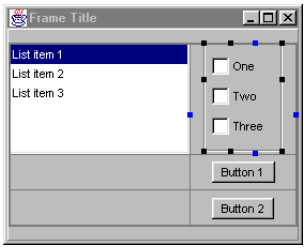
insets

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`.

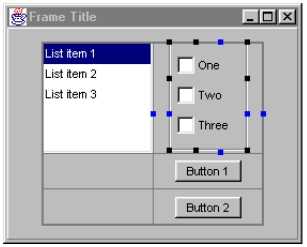
Setting inset values in the designer

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.

1. Right-click the component in the UI designer and choose Constraints to display the GridBagConstraints Editor.
2. 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:

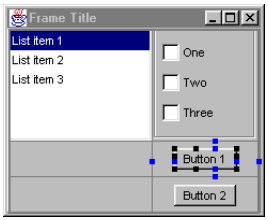
- If ipadx= 4, the component is 34 pixels wide.
- If ipady= 2, the component is 22 pixels high.

Setting the size of internal padding constraints in the designer

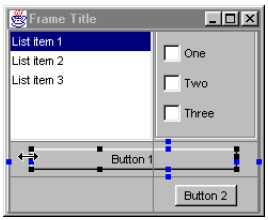
You can specify the size of a component's internal padding by clicking on any of the black sizing nibs at the edges of the component, and dragging with the mouse.

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 in the designer

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:

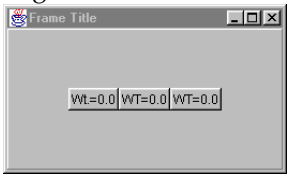
1. Activate the component's design context menu in one of two ways:
   - Right-click the component on the design surface.
   - Select the component in the component tree and press *Shift+F10*.
2. Choose Constraints.

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.

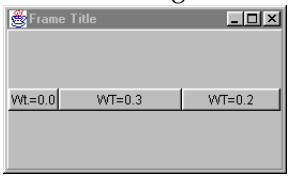
Important Because `weight` constraints can make the sizing behavior in the UI designer difficult to predict, setting these constraints should be the last step in designing a `GridBagLayout`.

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.

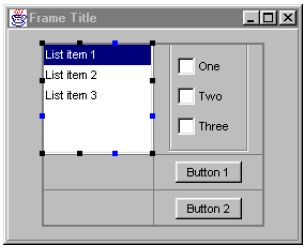


- You need to set both the `weight` and `fill` constraints for a component if you want it to grow. For example, if a component has a `weightx` constraint, but no horizontal `fill` constraint, then the extra space goes to the padding between the left and right edges of the component and the edges of the 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).

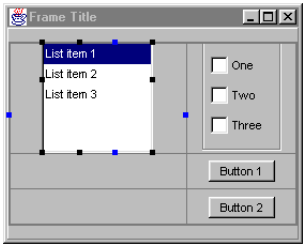
  The three pictures below demonstrate this.

  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`).

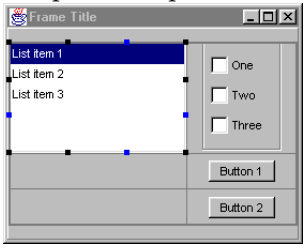


Next, a horizontal `weight` constraint of 1.0 is specified for the `ListControl`. Notice that as soon as one component is assigned any `weight`, the UI design is no longer centered in the panel. Since a horizontal `weight` constraint was used, the `GridBagLayout` manager takes the extra space in the 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.

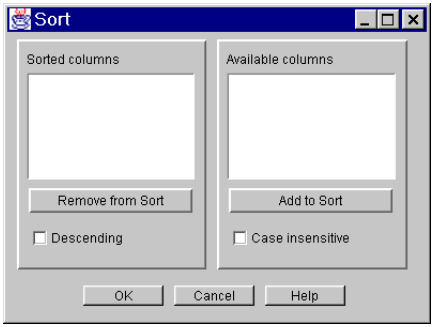


- If one component in a column has a `weightx` value, `GridBagLayout` gives the whole column that weight. Conversely, if one component in a row has a `weighty` value, the whole row is assigned that weight.

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));
```

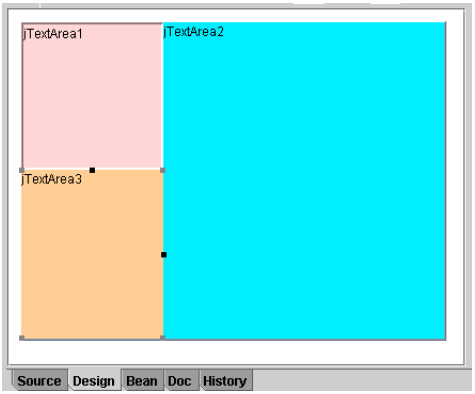
```
    jLabel2.setForeground(Color.black);
    jLabel2.setText("Sorted columns");
    jButton3.setText("Help");
    jButton4.setText("OK");
    jButton5.setText("Cancel");
    jPanel4.setLayout(gridLayout1);
    gridLayout1.setHgap(10);
    gridLayout1.setVgap(10);
    contentPane.add(jPanel1, BorderLayout.CENTER);
    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

This is a feature of JBuilder SE and Enterprise.

`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 8.10 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`	The name for this component (must be unique for all components in the container — as in `CardLayout`).
`String splitComponentName`	The name of the component from which space will be taken to make room for this component.

`String position`	The edge of the `splitComponentName` to which this component will be anchored.
	
	Valid values are:
	
	`PaneConstraints.TOP`
	This component will be above `splitComponentName`.
	
	`PaneConstraints.BOTTOM`
	This component will be below `splitComponentName`.
	
	`PaneConstraints.RIGHT`
	This component will be to the right of `splitComponentName`.
	
	`PaneConstraints.LEFT`
	This component will be to the left of `splitComponentName`.
	
	`PaneConstraints.ROOT`
	This component is the first component added.

`float proportion`	The proportion of `splitComponentName` 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 will always take 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`. The only choice is 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 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 Select a component on the component palette, and drop it into the `PaneLayout` container. This component will completely fill up the container until you add another component to split this one.

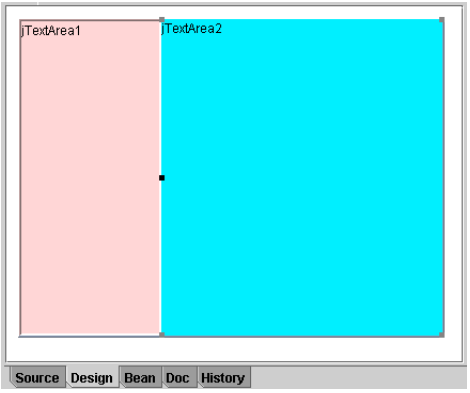
4 Select another component, and drag it across the first component to indicate where you want it to be anchored and how much space to take away from the first component.

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.

Note To make the edges of the components visible, we added a lowered bevel border to each component in the Inspector.

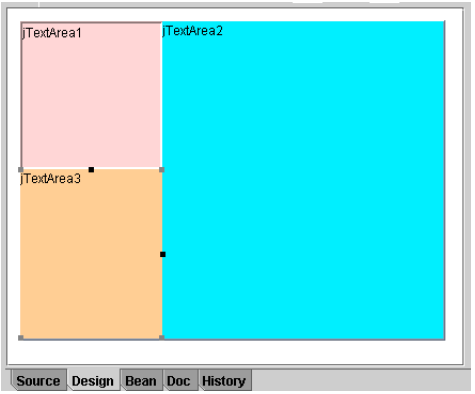
The layout manager will now split the space between the two components, giving the second component the area you defined, and giving the first component the rest of the frame or panel.



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.

5 To add a third component to the `PaneLayout`, draw it similarly to define its relative position to the other components.

For example, 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.



6 Use the same method to add subsequent components.

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, in JBuilder SE and Enterprise, `XYLayout` which make the initial design work easier.

Use XYLayout and null layout 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 8-11 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.

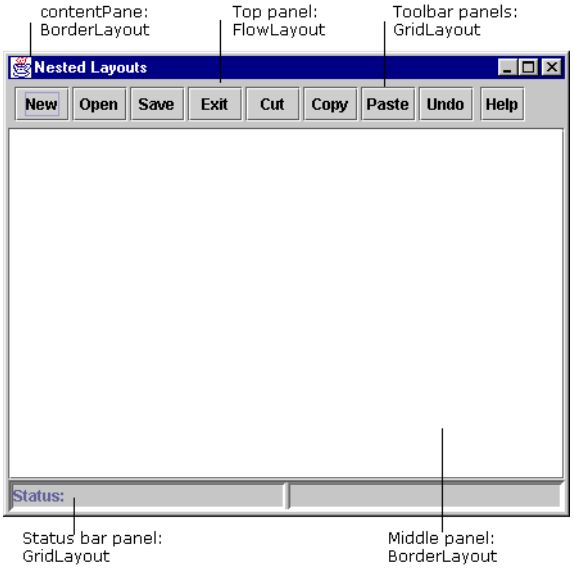
Even when you plan your UI first, you may discover that a particular layout you planned to use just doesn't work as you expected. This might mean reworking the design and using a different configuration of containers, components, and layouts. For this reason, you might want to copy the container file (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

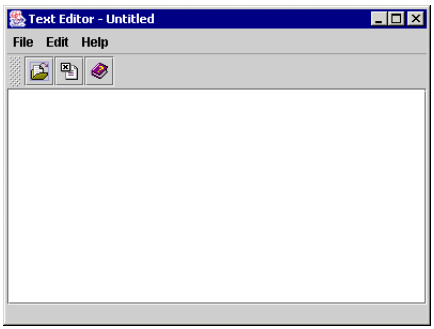
- Chapter 10, "Tutorial: Creating a UI with nested layouts," to work through the tutorial that builds this UI.
- Chapter 5, "Creating user interfaces," for more about using the designer.

Chapter

9

Tutorial: Building a Java text editor

This step-by-step tutorial uses JBuilder to build, test and run a Java application called "Text Editor". This application is a simple text editor capable of reading, writing, and editing text files.



This text editor will be able to set the text color and background color of the text editing region. In the JBuilder SE and Enterprise versions of the tutorial, it will also be able to set the text font.

The tutorial takes approximately two hours to complete.

What this tutorial demonstrates

Some steps in this tutorial are specific to JBuilder SE and Enterprise editions. This is noted at the top of those steps.

The Text Editor tutorial uses the Project and Application wizards to create a project and a set of visually designable files. Then it shows you how to use the visual design tools, modify the UI design, hook up events, and edit source code. It steps you through handling events for commonly used components and tasks, such as menu items, a toolbar, a text area, and

system events. It contains specific examples that show you how to do the following:

- Use the `JFileChooser` dialog box to allow the user to select a file.
- Read and write text from a text file and manipulate text with a `JTextArea`.
- Set foreground and background colors.
- Set the font using the dbSwing `FontChooser` dialog. This is a feature of JBuilder SE and Enterprise.
- Display information in a status bar and in the window caption.
- Add code manually to handle UI events.
- Have a menu item and a button execute the same code by putting the code in a new *helper* method called by both event handlers.
- Add a context menu to the `JTextArea` component.
- Keep track of the current filename and whether the file has been changed since the last save. Shows you how to handle the logic of this for File | New, File | Open, File | Save, File | SaveAs, editing, and exiting the file.
- Deploy the "Text Editor" application to a JAR file. This is a feature of JBuilder SE and Enterprise.

This tutorial contains code and text that you're expected to add. If you're using this tutorial onscreen, you can copy and paste code and blocks of text from the tutorial into the required fields.

Important If you're using a UNIX-based system and have installed JBuilder as root but are running as a regular user, copy the `Samples` tree to a directory in which you have full read/write permissions.

Sample code for this tutorial

To see the complete source for the TextEditor sample, open the sample project:

In Personal `<jbuilder>/samples/swing/SimpleTextEditor/SimpleTextEditor.jpx`

In SE and Enterprise `<jbuilder>/samples/TextEditor/TextEditor.jpx`

The `SimpleTextEditor` project doesn't include deployment code or code for setting background color. The `TextEditor` project does.

See also

- Chapter 1, "Visual design in JBuilder"
- "Using the AppBrowser" in *Introducing JBuilder*.

- "Building Java programs" in *Building Applications with JBuilder*.
- "Debugging Java programs" in *Building Applications with JBuilder*.

Step 1: Setting up

This tutorial creates a text editor that allows you to create, edit, and save files.

The functionality that allows you to create files is added after certain other functions are in place. We will test other functions first, such as opening and editing a file. Therefore, you need a file to work on.

1 Using your file manager, create a plain text file named `tester.txt`.

   Make sure that you have full read/write access and that it's a file that can be changed indiscriminately without harming any work.

2 Put text in it. You may copy and paste the text below:

```
Some text to use.

Text that will extend past one line in order to check that line wrap works
properly and displays as it should.
```

3 Save the file.

Next, create a project and the necessary files for building the text editor's user interface. We'll use the Project wizard to create the project, adjust some project settings manually, then use the Application wizard to create our files.

Creating the project

The Project wizard creates a new JBuilder project to work in.

1 Choose File | New Project to open the Project wizard.

2 Make the following changes in Step 1:

   - Name: `TextEditor`

Note By default, JBuilder uses this project name as the project's directory name and the package name for the classes inside the project.

   - Check the Generate Project Notes File option.

     When you check this option, the Project wizard creates an HTML file for project notes and adds it to the project.

   - If you have other projects open, uncheck the Add Project To Active Project Group option. This is a separate project.

3 Accept all other defaults in Step 1.

4. Click Next to go to Step 2 of the Project wizard.
5. Accept the default paths in Step 2.
6. Click Next to continue to Step 3 of the Project wizard.
7. Fill out the optional Javadoc fields.
   a. In the Title field, type `Text Editor Tutorial`.
   b. In the Description field, type `Tutorial demonstrating JBuilder's visual design features`.
   c. In the @author field, type your name.

   You may leave the other fields blank.

   This information is saved in the project HTML file. It's also used for Javadoc comments when you use the Generate Header Comments option offered in some of JBuilder's wizards, such as the Application and Class wizards.
8. Accept the other defaults on this page.
9. Press Finish to create the project.

   A project file and a project HTML file are added to the project and their nodes appear in the project pane.

See also

- "Managing paths" in *Building Applications with JBuilder*.
- "Creating and managing projects" in *Building Applications with JBuilder*.

Selecting the project's code style options

Now let's adjust the code style options. These options control how JBuilder writes event handler stubs and instantiation code. Event handling and instantiation are discussed in more detail later in this tutorial.

To change the code style options,

1. Right-click `TextEditor.jpx` in the project pane (upper left).
2. Choose Properties from the menu that appears.

   The Project Properties dialog box appears.
3. Click the Formatting tab in the Project Properties dialog box.
4. Click the Generated tab in the Formatting page.

   Here, we choose which style event handler to generate. JBuilder can use either anonymous inner classes or separate adapter classes. In this tutorial, we use separate adapter classes.

5 Look under Event Handling.

6 Select the Standard Adapter option.

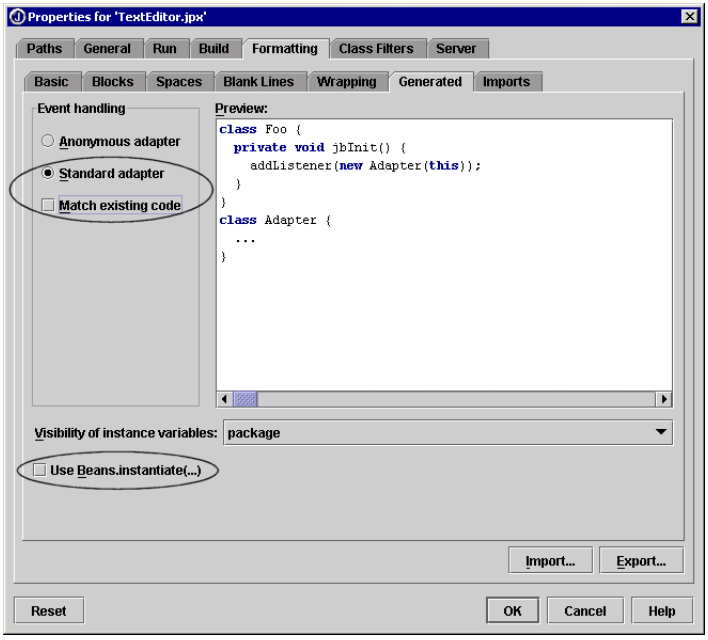
Note Regardless of which style event handler method you use, the code you put inside the method will be the same.

7 Deselect the Match Existing Code option. This helps make the code in this tutorial a little more predictable.

JBuilder gives you the option of instantiating objects using `Beans.instantiate()` instead of the keyword `new`. This tutorial uses `new`.

8 Make sure that the Use Beans.instantiate(...) option is *not* selected.

Your Generated page should now look like the image below. The options we're interested in are circled:



9 Click OK to close the Project Properties dialog.

See also

- "Choosing event handler style" on page 4-6.

Using the Application wizard

Now that we have a project, we need to populate it with visually designable files. Let's add the application files to the project.

1 Choose File | New to open the object gallery.

2. Select the General tab.
3. Double-click the Application icon to open the Application wizard.
4. Change the application class name in Step 1:
   - Class name: `TextEditClass`

   Accept the default package name.
5. Click Next to go to Step 2 of the Application wizard.
6. Change the name and title of the frame class:
   - Class: `TextEditFrame`
   - Title: `Text Editor`
7. Check all the options on Step 2. The wizard automatically generates code for the selected options.

   Notice what each option is as you check it off, so you know what to expect for generated code.
8. Click Next to go to Step 3, where JBuilder creates a default runtime configuration.

   Make sure this option is checked and accept the default name for the configuration.

   There are no base configurations available for this project because it's a new project.
9. Click the Finish button.

   Notice the `.java` and image files added to the project by the Application wizard.

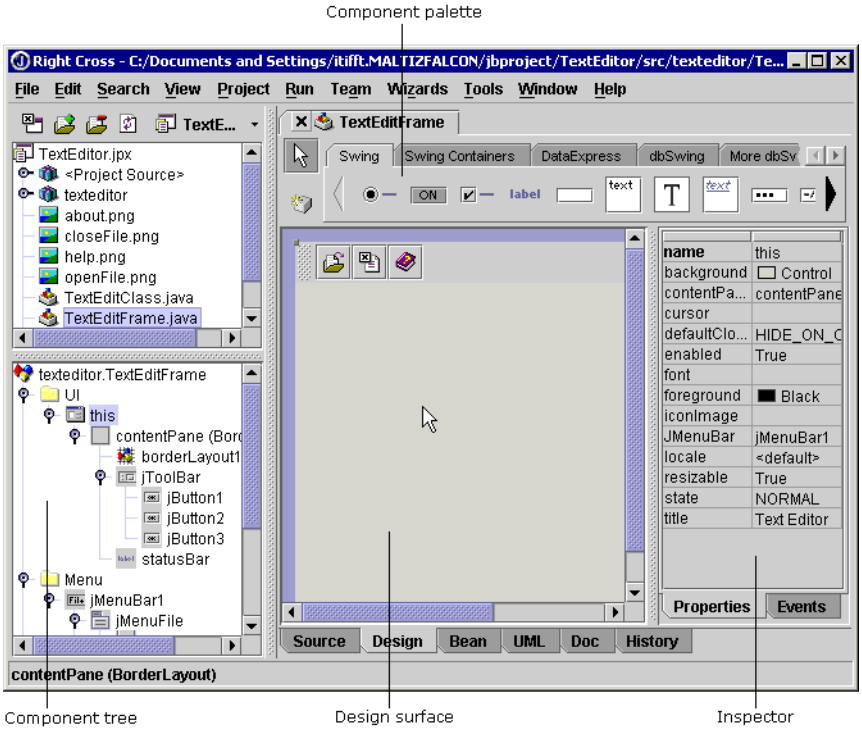
Note In JBuilder SE and Enterprise editions, an automatic source package node also appears in the project pane if the Automatic Source Packages option is enabled on the General page of the Project Properties dialog box (Project | Project Properties).

10. Save the project using File | Save Project "TextEditor.jpx".

Click the Design tab for the open file, `TextEditFrame.java`. The Design tab, located at the bottom of the AppBrowser window, opens the UI designer. Notice the changes in JBuilder's IDE:

- The UI designer is active in the content pane.
- The component tree appears in the structure pane, with `this` selected as the active component.
- The design surface appears in the content pane.
- The Inspector appears to the right of the design surface.

Figure 9.1 JBuilder in design view



Tip You can see which component your pointer is on on the design surface by looking in the status bar. This is helpful when you have a more complex UI design.

Tip If the design area is too narrow to see the entire UI in the AppBrowser, adjust the size of the AppBrowser panes either by dragging their borders with the mouse or by selecting Window | Select Browser Splitter, selecting the splitter bar you want to move, and using your arrow keys.

Suppressing automatic hiding of JFrame

By default, a `JFrame` will hide when you click its close box. This is not the behavior we want for this tutorial, because the Application wizard added an event handler to call `System.exit(0)` when the close button is pressed. Later we will be adding code in this handler to ask the user about saving the file on exit, and we do not want the window to automatically hide if the user says no.

To change the default behavior,

1. Select `this` in the component tree.
2. Click the Properties tab in the Inspector.

3 Select the `defaultCloseOperation` property value, `HIDE_ON_CLOSE`.

4 Choose DO_NOTHING_ON_CLOSE from the property's drop-down list.

Setting the look and feel

If you have changed the JBuilder look and feel from its default, then set up JBuilder so the designer will use the Metal Look & Feel. We'll use Metal for this tutorial because it looks pretty much the same across all supported platforms.

You can set the look and feel on the designer context menu or in the JBuilder IDE Options dialog box, but this doesn't have any effect on how your UI will look at runtime. To force a particular runtime look and feel, you have to set the look and feel explicitly in the `main()` method of the class that runs the application. In this case, the `main()` method is in `TextEditClass.java`.

By default, the Application wizard generates the following line of code in the `main()` method of the runnable class:

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

This means the runtime look and feel will be whatever the hosting system is using.

To specify Metal, do the following:

1 Double-click `TextEditClass.java` in the project pane to open the file in the editor.

2 Click `main(String[] args)` in the structure pane at the bottom left, or scroll down in the content pane until you find `public static void main(String[] args){`.

3 Highlight the `setLookAndFeel()` line of code and copy it to the line just below it.

4 Comment out the first version of this line of code with two forward slashes:

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

5 Change the argument of the new version to specify Metal Look And Feel:

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

6 Choose File | Save All to save the project and its files, then proceed to the next step.

It's a good idea to save frequently during this tutorial, for example, at the end of each step.

See also

- "Changing look and feel" on page 5-9.

Step 2: Adding a text area

In this step, you'll make a text area completely fill the UI frame between the menu bar at the top and the status bar at the bottom. To support this, the layout manager for the main UI container needs to use `BorderLayout`.

A `BorderLayout` container is divided into five areas: North, South, East, West, and Center. Each area can hold only one component, for a maximum of five components in the container. For this purpose, a panel containing multiple components is considered as one component. A North component clings to the top of the container, an East component to the left side, and so on. A component placed into the Center area completely fills the container space not occupied by any other areas containing components.

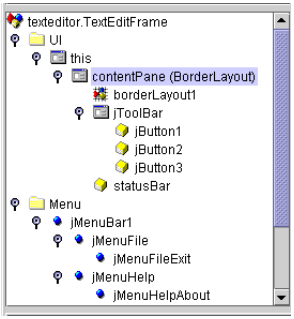
See also

- "BorderLayout" on page 8-15.

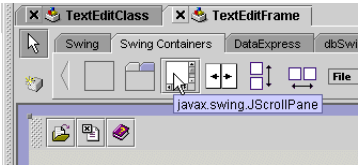
The Application wizard creates a `JFrame` component that's the main container for this UI. This `JFrame` component is the `this` component. `this` contains a `JPanel` object called `contentPane` which already uses `BorderLayout`. All you need to do now is add the components for the text area to the `contentPane`.

To do so, you'll add a scroll pane and then put a text area component inside it. The scroll pane provides the text area with scroll bars.

1. Select the `TextEditFrame` tab at the top of the editor.
2. Click the Design tab, if it's not already selected.
3. Click the `contentPane` component in the component tree to select it.



4 Click the Swing Containers tab on the component palette and select the `JScrollPane` component.



5 Click the center of the `contentPane` in the UI designer.

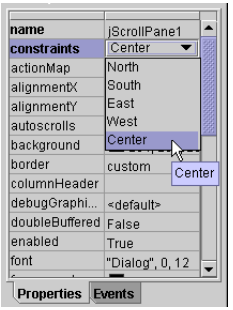
This drops the `JScrollPane` component into the `contentPane` panel and should give it a constraint of Center.

In this case, the toolbar occupies the North area (top), and the status bar occupies the South (bottom). Since no components are assigned to East and West, the scroll pane component occupies the Center area and expands to the left (West) and right (East) edges of the container.

If you miss, choose Edit | Undo and try again.

6 Select the new `jScrollPane1` component in the component tree.

7 Look at its `constraints` property value in the Inspector and verify that it is set to Center. If not, select Center from the property's drop-down list.



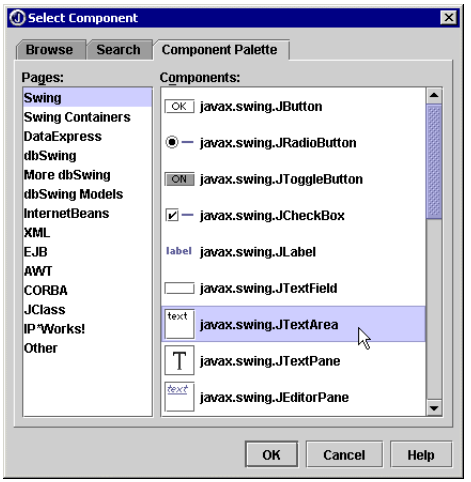
Let's add the text area, using a different way of adding components to the design. We'll use the Add Component dialog box.

1 Choose Edit | Add Component.

This brings up the Add Component dialog box.

2 On the Component Palette page, select Swing in the Pages area.

3 Select `javax.swing.JTextArea` in the Components area:



4 Click OK to close the dialog box.

The new component is added to `this`.

5 Choose Edit | Cut with the new component selected.

6 Select `jScrollPane1`.

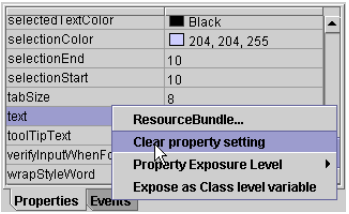
When you paste the text area component back in, this will be its parent.

7 Choose Edit | Paste.

The `jTextArea1` component appears below `jScrollPane1` in the component tree, and is nested inside it on the design surface.

There is default text inside the text pane. Let's take it away.

8 Right-click the `text` property in the Inspector and choose Clear Property Setting.



Finally, you need to set some properties of `jTextArea1` so it will wrap lines of text automatically and at word boundaries.

Tip The Inspector lists properties in alphabetical order.

In the Inspector, set these values for the following properties:

1. `background` = white
2. `lineWrap` = true
3. `wrapStyleWord` = true

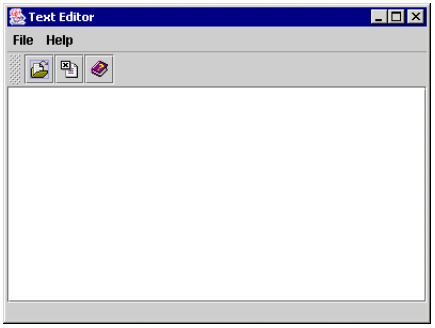
Now, compile your program then run it to see how it looks.

1. Choose Project | Make Project from the menu.

   This compiles all the files in the project. It generates a `TextEditClass.class` file and a `TextEditFrame.class` in the `classes` directory in the project directory. It should compile without any errors.

2. Click the Run button on the JBuilder toolbar, press *F9*, or choose Run | Run Project from the menu bar.

Your runtime UI should now look like this:

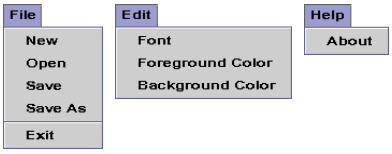


Notice that there are no scroll bars. This is because the `horizontalScrollBarPolicy` and `verticalScrollBarPolicy` properties for `jScrollPane1` are set to `AS_NEEDED` by default. If you want scroll bars to be visible all the time, you would need to change these property values to `ALWAYS`. We'll leave these properties as they are.

1. Choose File | Exit in the "Text Editor" application to close the runtime window.
2. To close the message pane, click the close button in the corner of the message tab:

Step 3: Creating the menus

In this step, you are going to create the following menus:



JBuilder Personal users, your tutorial omits the Edit|Font menu item.

You use the Menu designer to create and edit menus. We'll create new menu items, add a new menu, and insert a separator bar.

There are different ways to access these commands. This tutorial demonstrates many of them. Once you find a mode of command access you like, you may choose which mode to use in subsequent, similar commands.

1 Click the Design tab on `TextEditFrame.java`, if it's not already selected.

2 Open the Menu designer. Either double-click `jMenuFileExit` in the Menu folder in the component tree, or select it and press *Enter*.

   This switches the design surface to the Menu designer, with `jMenuFileExit` selected.

3 Insert a menu item using the Menu designer toolbar:

   a Click the Insert Menu Item button on the menu designer toolbar.

   b Type `New` directly in the new menu item location.

   c Press *Enter* to accept the new entry.

4 Insert a menu item using the Menu designer context menu:

   a Select the File menu item on the design surface.

      The File menu expands.

   b Right-click the Exit menu item.

      This displays a menu with all the Menu designer commands.

   c Choose Insert Menu Item from the Menu designer context menu.

   d In the Inspector's Properties page, select the text field.

   e Type `Open`.

   f Press *Enter* to accept the new entry and move down one line.

5 Insert two more menu items. Choose one of the above techniques to create the following menu items:

   a `Save`

   b `Save As`

6 Now, insert a bar between the Exit and Save As menu items:

   a Select the Exit menu item.

   b Click the Insert Separator toolbar button.

The File menu is now complete. Let's create a new menu, the Edit menu.

1 Right-click the Help item on the main menu bar and choose Insert Menu.

   This creates a new menu between the File and Help menus.

2 Type `Edit` as the name for this menu.

3 Press *Enter* to move down to the next blank entry. You don't need to press *Insert* here because there are no menu items on this menu after the current entry.

Note There's always a blank line at the bottom of a menu in the Menu designer. It's not a menu item, it's just a placeholder that JBuilder uses. You still need to use Insert Menu Item so that the new menu item is added above the placeholder.

Tip To delete an entry, select it and click the Delete toolbar button, or press the *Delete* key twice. The first press of the *Delete* key clears the text in the entry. The second press removes the entry from the menu.

4 Continue to build the Edit menu. Use your favorite technique for adding menu items. Add three items:

   a Font (JBuilder SE and Enterprise)

   b Foreground Color

   c Background Color

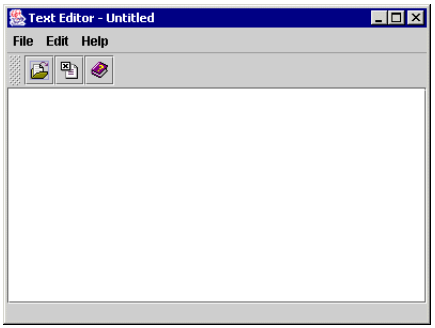
   If an entry is too wide for the edit area, the text automatically scrolls as you type. When you press *Enter*, the Menu designer adjusts the width of the menu to accommodate the longest item in the list.

5 Close the Menu designer by double-clicking any component in the UI folder of the component tree.

   This switches the view in the content pane back to the UI designer.

6 Save the file, then run the application.

Your UI should now look like this at runtime:



You should be able to play with the UI and type text in the text area, but the buttons won't do anything yet and only the File | Exit and Help | About menus will work.

You've created and populated the menus this UI requires. Now let's start to make them functional.

See also

- "Creating menus" on page 6-4.

Step 4: Adding a FontChooser dialog

JBuilder Personal users, skip Step 4 and 5. Go directly to Step 6. Ignore any directions in the rest of this tutorial pertaining to the Edit|Font menu item or the FontChooser dialog.

Let's begin hooking up the menu events, starting with the Edit | Font menu item. Once in place, this is going to bring up a `FontChooser` dialog.

Let's add a `FontChooser` dialog to `TextEditFrame.java` for the Font menu item to use:

1. Open `TextEditFrame.java` in the designer.
2. Choose Edit | Add Component.

   The Select Component dialog box appears.
3. In the Pages field, select More dbSwing. Click the Component Palette tab if it's not already open.

4. Select the `com.borland.dbswing.FontChooser` component from the Components field.
5. Click OK to add the font chooser to the design. It will appear as `fontChooser1` in the `Default` folder in the component tree.

You will only see the font chooser dialog component in the component tree, not in the UI designer.

Setting the dialog's frame and title properties

You need to set the `frame` property on this dialog component for it to work properly at runtime. The `frame` property must reference a `java.awt.Frame`, or descendant, before being shown. In this case, the frame you need to reference is 'this' frame (`TextEditFrame`). If you fail to do this, the dialog will not show, and an error message occurs at runtime. You can also set the `title` property so the dialog will have an appropriate caption.

To set the `frame` and `title` properties,

1. Select `fontChooser1` in the `Default` folder of the component tree.
2. Click the `frame` property value in the Inspector.
3. Select `this` from the drop-down list of values.
4. Click the `title` property value.
5. Type the word `Font` as its value.
6. Press *Enter*.

As a result of this, the following lines are added to the source code in the `jbInit()` method:

```
fontChooser1.setFrame(this);
fontChooser1.setTitle("Font");
```

Placing the `FontChooser` into the component tree and setting these properties creates code in your class that instantiates a `FontChooser` dialog for your class, sets its `title` to "Font", and sets its `frame` property to `this`. But this code won't display the dialog or make use of it in any way. The dialog has to be hooked up to the menu item first. That's done in the *event handler* for the Edit | Font menu item. Let's create that code now.

Creating an event to launch the FontChooser

Create an event for the Edit | Font menu item that will launch the `FontChooser`:

1. Select the Edit | Font menu item in the component tree. You can do this by choosing Edit | Font on the design surface. In the component tree, this component's name should be `jMenuItem5` and it should be under the second menu node, `jMenu1`.

Tip Don't worry if your Font menu item component has a different name. Just make sure you select the component for the Font menu item. The `text` property for this menu item in the Inspector says "Font".

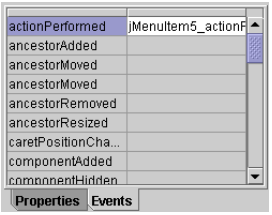
2. Click the Events tab in the Inspector.

   It displays an alphabetized list of all supported events for the selected component.

3 Click the value field (in the second column) for the `actionPerformed` event.

For menus, buttons, and many other Java UI components, `actionPerformed` is the main user event of interest, the one you should *hook* for responding to the user's interaction with that menu or button.

The name of the event handling method appears in the value field. If the method doesn't already exist, this will show the proposed default name for a new event handling method. For this new event handler, the suggested name is `jMenuItem5_actionPerformed`.



4 Double-click this event value, or press *Enter* to create the new event.

When an event handling method is new, double-clicking it in the Inspector generates an empty stub for the method in your source code.

Regardless of whether the method is new or already exists, the window focus will switch to the source code in the editor and position your cursor inside the event handling method.

For a new event handling method, as is the case here, you will see that there is no code yet in the body of the method.

5 Type the following line of code inside the body of this new empty method between the open and close curly braces:

```
fontChooser1.showDialog();
```

Your method should now look like this:

```
void jMenuItem5_actionPerformed(ActionEvent e) {
    fontChooser1.showDialog();
  }
```

Tip To increase viewing area in the content pane, either move the splitters at the borders using your mouse or select Window | Select Splitter | Project/Content and use your arrow keys to move the splitter.

6 Now save and run your application. The Edit | Font menu item should open a `FontChooser` dialog. If not, check that you set its `frame` property to `this`.

7 Close the "Text Editor" application and close the JBuilder message pane.

Nothing happens yet when you try to change the font. This is because the application isn't using the results from the `FontChooser` to change the text in the text area. Let's make it do that next.

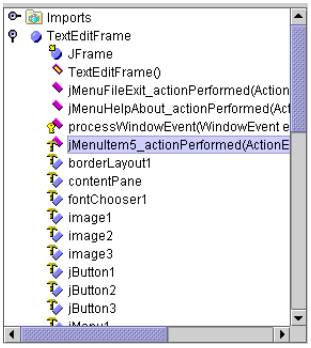
Step 5: Attaching a menu item event to the FontChooser

JBuilder Personal users, skip this step and go to Step 6. Ignore any directions in the rest of this tutorial pertaining to the Edit|Font menu item or FontChooser dialog.

In this step, we hook the `FontChooser` dialog to the text area component, making it possible for the text area to use the font chooser.

1 Click the Source tab and go to the Font menu item event handling method (`jMenuItem5_actionPerformed(ActionEvent e)`) that you just created, if you're not there already.

Tip To quickly locate this method in the source code, click the following node in the structure pane (bottom left of the AppBrowser). Note that the order of the elements in your structure pane might not appear exactly as they do here; the order depends on the setting of the Structure Order options on the Java Structure page of the Structure View Properties dialog box.



2 Insert the following code into your Font menu item (`jMenuItem5`) event handling method between the opening and closing curly braces, being sure to replace the old `fontChooser1.showDialog();` code:

```
// Handling the "Edit Font" menu item

// Pick up the existing font from the text area and put it into the FontChooser
// before showing the FontChooser, so that we are editing the current font.
fontChooser1.setSelectedFont(jTextArea1.getFont());

// Test the return value of showDialog() to see if the user pressed OK.
// Obtain the new Font from the FontChooser.
if (fontChooser1.showDialog()) {

  // Set the font of jTextArea1 to the font
  // the user selected before pressing the OK button.
  jTextArea1.setFont(fontChooser1.getSelectedFont());
}
```

The entire method should now look like this:

```
void jMenuItem5_actionPerformed(ActionEvent e) {
  // Handling the "Edit Font" menu item

  // Pick up the existing font from the text area and put it
  // into the FontChooser before showing the FontChooser,
  // so that we are editing the current font.
  fontChooser1.setSelectedFont(jTextArea1.getFont());

  // Test the return value of showDialog() to see if the user
  // pressed OK. Obtain the new Font from the FontChooser.
  if (fontChooser1.showDialog()) {

    // Set the font of jTextArea1 to the font
    // the user selected before pressing the OK button.
    jTextArea1.setFont(fontChooser1.getSelectedFont());
  }
}
```

Tip To save typing, you can copy and paste the code example above from the Help Viewer to your source code by doing the following:

a Select the code to copy in the Help Viewer. In this example, highlight the entire event handling method. Be sure to check your curly braces, so you wind up with the right number.

b Choose Edit | Copy on the Help Viewer menu or use your keymapping's keystroke shortcut.

c Click the Source tab to switch to the editor in the AppBrowser.

d Highlight the code you want to replace. In this example, highlight the entire event handling method in your source code.

Warning Be careful where you paste. Don't remove an important curly brace, such as the closing one for the class definition.

e Choose Edit | Paste from the JBuilder main menu or use the appropriate keyboard shortcut.

f Check the indenting level of the inserted code and adjust to match your code. Indent a block by selecting the text and pressing the `Tab` key.

3 Save and run the application and type some text in the text area.

4 Select the text and use the Edit | Font menu item to change the text's font.

5 Close the "Text Editor" application and close the JBuilder message pane.

This application changes the font for the entire text area, not just selected text. It doesn't persist the new font settings, so when you close and reopen the application, the default font is used again. We aren't going to enter code to enable these features in this tutorial, but you could do that as an independent exercise after you complete the tutorial.

Step 6: Attaching menu item events to JColorChooser

Now let's create Edit | Foreground Color and Edit | Background Color menu events and connect them to a `javax.swing.JColorChooser` dialog.

Since you don't need to change any of the properties for `JColorChooser` in this application, there's no need to add the component to the UI codebase. You can just call it directly from a menu item's `actionPerformed()` event handler as follows:

1. Switch back to the Menu designer for `TextEditFrame.java`.
2. Select the second menu item in the component tree under Edit (`jMenuItem6`) which has "Foreground Color" in its `actionCommand` property on the Properties page of the Inspector.
3. Click the Events tab in the Inspector and double-click the `actionPerformed()` event to create the following event handler:

```
void jMenuItem6_actionPerformed(ActionEvent e) {
}
```

4. Insert the following code into the stub of the event handler (including comments if you wish):

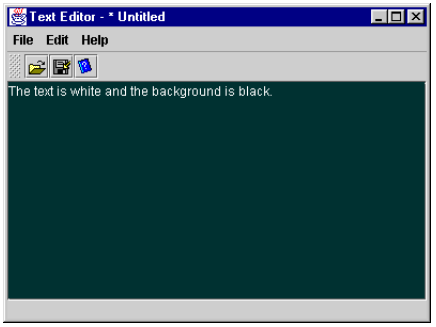
```
// Handle the "Foreground Color" menu item
Color color = JColorChooser.showDialog(this,"Foreground Color",
    jTextArea1.getForeground());
if (color != null) {
  jTextArea1.setForeground(color);
}
```

5. Switch back to the Menu designer.
6. Select the third menu item in the component tree under Edit (`jMenuItem7`), which should have "Background Color" in its `actionCommand` property. Create an `actionPerformed()` event for it as you did for `jMenuItem6`.
7. Insert the following code into the `actionPerformed()` event for `jMenuItem7`:

```
// Handle the "Background Color" menu item
Color color = JColorChooser.showDialog(this,"Background Color",
    jTextArea1.getBackground());
if (color != null) {
  jTextArea1.setBackground(color);
}
```

8. Save your file, then compile and run your application.

Type in text and play around with the foreground and background colors. Here is what it looks like if you set the foreground to white and the background to black:



9 Close the "Text Editor" application and close the JBuilder message pane.

Step 7: Adding a menu event handler to clear the text area

Let's hook up the File | New menu item to an event handler that clears the old text out of the text area when you open a new file.

1 Switch back to the Menu designer.

2 Select the File | New menu item in the component tree (probably `jMenuItem1`).

3 Create an actionPerformed() method, as you now know how to do.

4 Insert the following code into it:

```
// Handle the File|New menu item.
// Clears the text of the text area.
jTextArea1.setText("");
```

5 Save and run the application, type something into the text area, then see what happens when you choose File | New. It should erase the contents.

Notice that it doesn't ask you if you want to save your file first. To handle that, you need to set up infrastructure for reading and writing text files, for tracking whether the file has changed and needs saving, and so on. Let's begin the file support in the next step.

6 Close the "Text Editor" application and close the JBuilder message pane.

Step 8: Adding a file chooser dialog

Let's hook up the File | Open menu item to an event handler that presents the user with a `JFileChooser` (file open dialog) for text files. If the user selects a file and clicks the OK button, then the event handler opens that text file and puts the text into the `JTextArea`.

1. Switch back to the designer and select the `JFileChooser` component from the Swing Containers page of the palette.
2. Click the `UI` folder in the component tree to drop the component into the UI designer. (If you click on the design surface, the component will be dropped into the wrong section of the tree.)
3. Select the File | Open menu item in the component tree (probably `jMenuItem2`).
4. Create an `actionPerformed()` event and insert the following code:

```
//Handle the File|Open menu item.
// Use the OPEN version of the dialog, test return for Approve/Cancel
if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {

 // Display the name of the opened directory+file in the statusBar.
 statusBar.setText("Opened "+jFileChooser1.getSelectedFile().getPath());

 // Code will need to go here to actually load text
 // from file into TextArea.
}
```

5. Save and run the application.
6. Using the File | Open menu, select a file and click OK.

   You should see the complete directory and file name displayed in the status line at the bottom of the window. However, no text appears in the text area. We'll take care of that in the next step.
7. Close the "Text Editor" application before continuing.

Internationalizing Swing components

JBuilder Personal users skip this step and go to Step 9.

Imagine that we're localizing this application to run in several languages. This means we need to add a line of code so the Swing components, `JFileChooser` and `JColorChooser`, will appear in the language which the user runs the application in.

1. Add the following line of code to the `TextEditFrame` class in `TextEditFrame.java`:

```
IntlSwingSupport intlSwingSupport1 = new IntlSwingSupport();
```

Your code now looks like this:

```
public class TextEditFrame extends JFrame {
  IntlSwingSupport intlSwingSupport1 = new IntlSwingSupport();
  JPanel contentPane;
  JMenuBar menuBar1 = new JMenuBar();
  JMenu menuFile = new JMenu();
  ...
}
```

Note In this tutorial, the import statement `import com.borland.dbswing.*;` was added automatically when you added the dbSwing `FontChooser` component. In other situations, you could compile the file then use Optimize Imports to add the necessary import statements automatically.

Now, when you run your application in other languages, the `JFileChooser` and `JColorChooser` will appear in the appropriate language.

2 Save the application.

See also

- "Internationalizing programs with JBuilder" in *Building Applications with JBuilder*.
- "Adding and configuring libraries" in *Building Applications with JBuilder*.
- "Optimize Imports" in *Building Applications with JBuilder*.

Step 9: Adding code to read text from a file

In this step, we'll add the code that actually reads text from the user-selected file into the `JTextArea`. This involves adding a method to `TextEditFrame.java` and adjusting the event handler that calls the method.

First, add a new method to your class to perform the actual open file operation. We'll call this method `openFile()`.

1 Switch to the editor in `TextEditFrame.java`.

2 Add the following import to the list of imports at the top of the file:

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

3 Insert the following `openFile()` method.

You can put this method anywhere in your class (outside of other methods). A good place for it is just after the code for the `jbInit()` method, just before the `jMenuFileExit_actionPerformed()` event.

```
// Open named file; read text from file into jTextArea1; report to
statusBar.
void openFile(String fileName) {
  try {
    // Open a file of the given name.
    File file = new File(fileName);

    // Get the size of the opened file.
    int size = (int)file.length();

    // Set to zero a counter for counting the number of
    // characters that have been read from the file.
    int chars_read = 0;

    // Create an input reader based on the file, so we can read its data.
    // FileReader handles international character encoding conversions.
    FileReader in = new FileReader(file);

    // Create a character array of the size of the file,
    // to use as a data buffer, into which we will read
    // the text data.
    char[] data = new char[size];

    // Read all available characters into the buffer.
    while(in.ready()) {
      // Increment the count for each character read,
      // and accumulate them in the data buffer.
      chars_read += in.read(data, chars_read, size - chars_read);
    }

    in.close();

    // Create a temporary string containing the data,
    // and set the string into the JTextArea.
    jTextArea1.setText(new String(data, 0, chars_read));

    // Display the name of the opened directory+file in the statusBar.
    statusBar.setText("Opened "+fileName);
  }

  catch (IOException e) {
    statusBar.setText("Error opening "+fileName);
  }
}
```

4 Click the File | Open event handler in the structure pane to locate it in the source code. Its name will be `jMenuItem2_actionPerformed(ActionEvent)` if the File | Open menu item component name is `jMenuItem2`.

5 Replace the code in the File | Open event handler `if()` statement that previously said:

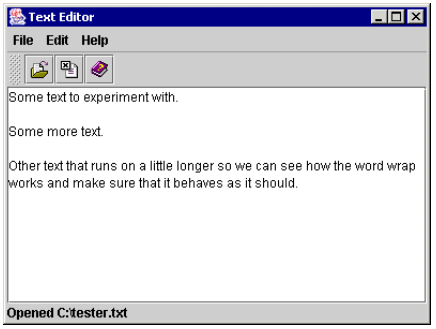
```
// Display the name of the opened directory+file in the statusBar.
statusBar.setText("Opened "+jFileChooser1.getSelectedFile().getPath());

// Code will need to go here to actually load text
// from file into JTextArea.
```

with this new `openFile()` method instead, using the concatenated directory and file name:

```
// Call openFile to attempt to load the text from file into JTextArea
openFile(jFileChooser1.getSelectedFile().getPath());
//repaints menu after item is selected
this.repaint();
```

6 Save and run your program and open `tester.txt` in your editor.

The correct text file should appear in the text editor:



7 Close the "Text Editor" application and close the JBuilder message pane.

Step 10: Adding code to menu items for saving a file

We need code that writes the file back out to disk when File | Save and File | Save As are used. The program needs to know whether the file being saved is a new file or whether an existing file has been modified. When a file has been modified since the last save, it's called a *dirty* file.

We'll add a `String` instance variable to hold the name of the file that was opened and add methods for checking if the file is dirty and writing the text back out.

1. Click `jFileChooser1` in the structure pane. This will take you to the last entry in the list of instance variable declarations (since `jFileChooser1` was the last declaration made).
2. Add the following declarations to the end of the list after `jFileChooser1`:

```
String currFileName = null;  // Full path and filename.
                             // null means new/untitled.
boolean dirty = false;  // false means the file was not modified initially.
```

3. Click the `openFile(String fileName)` method in the structure pane to quickly locate it in the source code. Place the cursor in that method after the following line that reads the file into the `JTextArea`:

```
jTextArea1.setText(new String(data, 0, chars_read));
```

4. Insert the following code there:

```
// Cache the currently opened filename for use at save time...
this.currFileName = fileName;
// ...and mark the edit session as being clean
this.dirty = false;
```

5. Create the following `saveFile()` method that you can call from the File | Save event handler. You can place it just after the `openFile()` method block. This method also writes the file name to the status bar upon saving.

```
// Save current file; handle not yet having a filename; report to statusBar.
boolean saveFile() {

  // Handle the case where we don't have a file name yet.
  if (currFileName == null) {
    return saveAsFile();
  }

  try {
    // Open a file of the current name.
    File file = new File (currFileName);

    // Create an output writer that will write to that file.
    // FileWriter handles international characters encoding conversions.
    FileWriter out = new FileWriter(file);
    String text = jTextArea1.getText();
    out.write(text);
    out.close();
    this.dirty = false;

    // Display the name of the saved directory+file in the statusBar.
    statusBar.setText("Saved to " + currFileName);
    return true;
  }
```

```
    catch (IOException e) {
      statusBar.setText("Error saving "+ currFileName);
    }
    return false;
  }
```

After you've created the above code, you'll see an error message in the structure pane: "Method saveAsFile() not found in class texteditor.TextEditFrame." We'll take care of that now.

6 Create the following `saveAsFile()` method. It's called from `saveFile()` when a new file is saved. It will also be used from the File | Save As menu item, which we'll handle later. Put the following code right after the `saveFile()` method block:

```
// Save current file, asking user for new destination name.
// Report to statusBar.
boolean saveAsFile() {
  // Use the SAVE version of the dialog, test return for Approve/Cancel
  if (JFileChooser.APPROVE_OPTION == jFileChooser1.showSaveDialog(this)) {
    // Set the current file name to the user's selection,
    // then do a regular saveFile
    currFileName = jFileChooser1.getSelectedFile().getPath();
    //repaints menu after item is selected
    this.repaint();
    return saveFile();
  }
  else {
    this.repaint();
    return false;
  }
}
```

7 Switch back to the Menu designer and create an `actionPerformed()` event handler for the File | Save menu item (probably `jMenuItem3`). Insert the following code:

```
//Handle the File|Save menu item.
saveFile();
```

8 Create an `actionPerformed()` event handler for the File | Save As menu item (`jMenuItem4`) and insert the following code:

```
//Handle the File|Save As menu item.
saveAsFile();
```

9 Save, compile, and run the program.

10 Use the application to open `tester.txt`, make changes in the file, and save the changes. Make more changes and save `tester.txt` as `tester_1.txt`.

11 Close the "Text Editor" application and close the JBuilder message pane.

Step 11: Adding code to test if a file has been modified

The program needs to keep track of whether a file has been modified since being created, opened, or saved, so the program can appropriately prompt the user when the file should be saved before closing the file or exiting the program. To do this, we'll add the `boolean` variable called `dirty`, which we've already referred to in previous code.

1. Click the following File | New event-handling method in the structure pane: `jMenuItem1_actionPerformed(ActionEvent e)`
2. Add the following code to the end of this method to clear the `dirty` and `currFileName` variables. Place it immediately after the line `jTextArea1.setText("");` and before the closing curly brace.

```
// Clear the current filename and reset the file to clean.
currFileName = null;
dirty = false;
```

   You'll use the `JOptionPane` dialog to display a confirmation message box to find out from the user whether to save a dirty file before abandoning it when doing a File | Open, File | New, or File | Exit. This dialog is invoked by calling a class method in `JOptionPane`, so you don't need to add a `JOptionPane` component to your program.

3. Add the following `okToAbandon()` method to the source code. You can put this new method right after the `saveAsFile()` method block:

```
// Check if file is dirty.
// If so, prompt for save/don't save/cancel save decision.
boolean okToAbandon() {
  int value =  JOptionPane.showConfirmDialog(this, "Save changes?",
      "Text Edit", JOptionPane.YES_NO_CANCEL_OPTION) ;

  switch (value) {
     case JOptionPane.YES_OPTION:
       // Yes, please save changes
       return saveFile();
     case JOptionPane.NO_OPTION:
       // No, abandon edits; that is, return true without saving
       return true;
     case JOptionPane.CANCEL_OPTION:
     default:
       // Cancel the dialog without saving or closing
       return false;
  }
}
```

   This method is not yet complete, but we'll finish it later.

   This method will be called whenever the user chooses File | New, File | Open, or File | Exit. Its purpose is to test to see if the text needs to be saved. If it is dirty, this method uses a yes/no/cancel message dialog to ask the user whether to save the file.

This method also calls `saveFile()` if the user clicks the Yes button. When the method returns the `boolean` value true, it indicates it is OK to abandon this file because it was clean or the user clicked the Yes or No button. If the return value is false, it means the user clicked Cancel. The code that will actually check to see if the file has changed will be added in a later step.

For now, this method always treats the file as dirty, even if no change has been made to the text. Later you will add a method to set the `dirty` variable to `true` when the user types in the text area, and you will add code to the top of `okToAbandon()` to test the `dirty` variable.

4 Place calls to this `okToAbandon()` method at the top of your File | New and File | Open event handlers, as well as in the wizard-generated File | Exit event handler. In each case, test the value returned by `okToAbandon()` and only perform the operation if the value returned is `true`.

Tip To find these event handlers quickly, click them in the structure pane. You can also search in the structure pane by moving focus to the structure pane and typing.

The following are the modified event handlers:

- For **File | New**, put a new `if` statement in the method body so that code will only be executed if `okToAbandon()` returns `true`. The modified method should now look like this:

```
void jMenuItem1_actionPerformed(ActionEvent e) {
  // Handle the File|New menu item.
  if (okToAbandon()) {
     // clears the text of the TextArea
     jTextArea1.setText("");
    // clear the current filename and set the file as clean:
    currFileName = null;
    dirty = false;
  }
}
```

- For **File | Open**, put an `if` statement in the method for when `okToAbandon()` returns `true`, *and* add code to return right away from the method if `okToAbandon()` returns `false`.

  The modified method should now look like this:

```
void jMenuItem2_actionPerformed(ActionEvent e) {
  //Handle the File|Open menu item.
  if (!okToAbandon()) {
    return;
  }
  // Use the OPEN version of the dialog, test return for Approve/Cancel
  if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this))
{
     // Call openFile to attempt to load the text from file into TextArea
    openFile(jFileChooser1.getSelectedFile().getPath());
  }
```

```
    this.repaint();
  }
```

- For **File | Exit**, put a test for `okToAbandon()` around the line of code that exits the application. The modified method should now look like this:

```
//File | Exit action performed
public void jMenuFileExit_actionPerformed(ActionEvent e) {
  if (okToAbandon()) {
    System.exit(0);
  }
}
```

Each of these menu event handling methods now does its task if `okToAbandon()` returns `true`.

5 Save and run the program and try opening, editing, and saving `tester.txt` and `tester_1.txt`.

Remember that `okToAbandon()` isn't completed yet. Right now, it always acts like the file is dirty. The result is that, for now, the confirmation message box always comes up when you choose File | New, File | Open, or File | Exit, even if the text hasn't been changed. If the file hasn't been changed, click Cancel to close the dialog and continue executing the command.

6 Close the "Text Editor" application and close the JBuilder message pane.

Step 12: Activating the toolbar buttons

When we generated files with the Application wizard, we checked the Generate Toolbar option. This made JBuilder generate code for a `JToolBar` and populate it with three `JButton` components that already display icons. All we have to do is specify the text for each button's label and tool tip and create an `actionPerformed()` event for each button to call an appropriate event-handling method.

Specifying button tool tip text

To specify tool tips for the buttons,

1 Switch to the UI designer.

2 Select `jButton1` under `jToolBar` in the component tree.

3 Click the Properties tab in the Inspector.

4 Click the `toolTipText` property to highlight its entry.

5 Type `Open File`, if it doesn't already say that, and press *Enter*.

6 Repeat this process for `jButton2` and `jButton3`, using the following text:

- Type `Save File` for `jButton2`.
- Type `About` for `jButton3`.

Creating the button events

Up until now, we have created event handlers using the Inspector. Let's use a shortcut to create the button events.

Many components define a default event in their BeanInfo class. For example, a button defines `actionPerformed()` as its default event. To generate an event handler quickly for this default event, double-click the component in the design surface.

Using this shortcut, create events for the buttons as follows:

1 Double-click `jButton1` on the design surface. This switches to the editor and places your cursor inside the new `jButton1_actionPerformed(ActionEvent e)` event for the Open button.

2 Enter the following code to call the `fileOpen()` method:

```
//Handle toolbar Open button
fileOpen();
```

3 Create a `jButton2_actionPerformed(ActionEvent e)` event for `jButton2` and call `saveFile()` from it:

```
//Handle toolbar Save button
saveFile();
```

4 Create a `jButton3_actionPerformed(ActionEvent e)` event for `jButton3`, and call `helpAbout()` from it:

```
//Handle toolbar About button
  helpAbout();
```

Notice that the code in the `jButton1` and `jButton3` event-handlers make calls to methods which don't exist yet: `fileOpen()` and `helpAbout()`. Let's create those methods now.

Creating a fileOpen() method

The `fileOpen()` method performs the operations that are currently in your File | Open menu item handling method. However, since you need to perform the same operations when the Open button is pressed, you'll create the `fileOpen()` method so you can have just one copy of that code, and call it from both the File | Open menu and the Open button.

1 Create a `fileOpen()` method stub. You can put this method just above the `openFile(String fileName)` method. The stub should look like this:

```
// Handle the File|Open menu or button, invoking
// okToAbandon and openFile as needed.
 void fileOpen() {
}
```

2 Go to your existing File | Open event handler, `jMenuItem2_actionPerformed()`. Select all the code between the first comment and the last closing curly brace in `jMenuItem2_actionPerformed()`. The code selected should be:

```
if (okToAbandon()) {
  return;
}

// Use the OPEN version of the dialog, test return for Approve/Cancel
if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {
  // Call openFile to attempt to load the text from file into TextArea
  openFile(jFileChooser1.getSelectedFile().getPath());
}
this.repaint();
```

3 Cut this code out of the `jMenuItem2_actionPerformed()` block and paste it into the new `fileOpen()` method stub.

Here is what the completed `fileOpen()` method looks like:

```
// Handle the File|Open menu or button, invoking okToAbandon and openFile
// as needed.
void fileOpen() {
  if (!okToAbandon()) {
    return;
  }
  // Use the OPEN version of the dialog, test return for Approve/Cancel
  if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {
    // Call openFile to attempt to load the text from file into TextArea
    openFile(jFileChooser1.getSelectedFile().getPath());
  }
  this.repaint();
}
```

4 Now, call `fileOpen()` from the File | Open menu item event handler. When you add the call to the empty stub, the event handler looks like this:

```
void jMenuItem2_actionPerformed(ActionEvent e) {
  // Handle the File|Open menu item.
  fileOpen();
}
```

Creating a helpAbout() method

Now do a similar thing for the Help | About menu item and the About button. Gather the code that is currently in the Help | About event handler into a new `helpAbout()` method and call it from both the menu and button event handlers.

1 Place this `helpAbout()` method stub in your code just before the `fileOpen()` method:

```
// Display the About box.
void helpAbout() {
}
```

2 Cut the following code out of `jMenuHelpAbout_actionPerformed()` and paste it into the new `helpAbout()` method stub:

```
TextEditFrame_AboutBox dlg = new TextEditFrame_AboutBox(this);
Dimension dlgSize = dlg.getPreferredSize();
Dimension frmSize = getSize();
Point loc = getLocation();
dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x,
    (frmSize.height - dlgSize.height) / 2 + loc.y);
dlg.setModal(true);
dlg.show();
```

3 Insert the call `helpAbout();` into `jMenuHelpAbout_actionPerformed()` so the method looks like this:

```
//Help | About action performed
public void jMenuHelpAbout_actionPerformed(ActionEvent e) {
  helpAbout();
}
```

4 Now, save and run the application. Try the Open, Save, and About buttons. Compare them with the File | Open, File | Save, and Help | About menu items.

5 Close the "Text Editor" application and close the JBuilder message pane.

Step 13: Hooking up event handling to the text area

Now, hook up the event handling to the `JTextArea` so your program can mark the file as dirty whenever the user modifies the file.

To understand what we're going to do, remember that Swing is architected so that the UI is completely separate from the data being represented in it. UI components have a set of classes all to themselves, and information to be represented has a set of classes that hold and manipulate the data. The data-holding classes are called *models*. Lists use `ListModel`, tables use `TableModel`, and documents use `Document`.

We'll add a Swing `DocumentListener` to the `jTextArea`'s `DocumentModel` and check for events that insert, remove, or change things in the file.

1 Switch to design mode and select `jTextArea1`.

2 Right-click the `document` property in the left column of the Inspector.

3 Choose Expose As Class Level Variable from the context menu.

A `document1` object is placed in the `Default` folder of the component tree, where we can now set its properties and events.

4. Select `document1` in the component tree, then switch to the Events tab in the Inspector.
5. Create a `changedUpdate()` event by double-clicking the event's value field.

   Look inside the `jbInit()` method for this new `DocumentListener`:

   ```
   document1.addDocumentListener(new
   TextEditFrame_document1_documentAdapter(this));
   ```

Tip Quickly search in the editor using the Find tool in the toolbar.

6. Insert the following code into the `document1_changedUpdate(DocumentEvent e)` event stub you created:

   ```
   dirty = true;
   ```

7. Return to the designer, select `document1`, and create two more events from the Inspector for `document1`: `insertUpdate()` and `removeUpdate()`. Insert the same line of code in these events that you used in the `changedUpdate()` event.

   This will make sure that any character typed in the text area will force the `dirty` flag to true.

8. Add the following three lines to the top of the `okToAbandon()` method so that now it will really be testing the `dirty` value:

   ```
   if (!dirty) {
     return true;
   }
   ```

The `okToAbandon()` method should now look like this:

```
// Check if file is dirty.
// If so, prompt for save/don't save/cancel save decision.
boolean okToAbandon() {
  if (!dirty) {
    return true;
  }
  int value =  JOptionPane.showConfirmDialog(this, "Save changes?",
                                        "Text Edit",
JOptionPane.YES_NO_CANCEL_OPTION) ;
    switch (value) {
    case JOptionPane.YES_OPTION:

      // Yes, please save changes
      return saveFile();
    case JOptionPane.NO_OPTION:

      // No, abandon edits; that is, return true without saving
      return true;
    case JOptionPane.CANCEL_OPTION:
    default:

      // Cancel the dialog without saving or closing
     return false;
  }
}
```

9 Save your work, run the program, and test to see that dirty and clean states of the file work properly. The Save Changes prompt should not appear if you use File | New, File | Open, or File | Exit on a clean file, but should appear when you use these commands on a dirty file.

10 Close the "Text Editor" application and close the JBuilder message pane.

Step 14: Adding a context menu to the text area

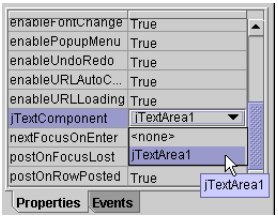
This step is for JBuilder SE and Enterprise only. Personal users skip this step and go to Step 15.

The `DBTextDataBinder` component adds a *context menu* to Swing text components for performing simple editing tasks such as cutting, copying, or pasting clipboard data. A context menu is a menu that's accessed by right-clicking a UI object, and contains only commands that are pertinent in that object.

`DBTextDataBinder` also has built-in actions to load and save files into a `JTextArea`, but they don't allow you to retrieve the file name loaded or saved. Unfortunately, we want this information because this application displays the file name in the status bar. Therefore, we are going to add a

DBTextDataBinder, bind it to jTextArea1, and then suppress its Open and Save actions.

1. Click the Design tab and select the DBTextDataBinder component on the dbSwing Models tab of the component palette.
2. Drop it anywhere in the designer or on the component tree. It is placed in the Data Access folder in the tree as dBTextDataBinder1.
3. Select dBTextDataBinder1 in the component tree, and then open its jTextComponent property value list in the Inspector.
4. Choose jTextArea1 from the drop-down list.



This binds dBTextDataBinder1 to jTextArea1 by placing the following line of code in the jbInit() method.

```
dBTextDataBinder1.setJTextComponent(jTextArea1);
```

5. Select the enableFileLoading property for dBTextDataBinder1 and set its value to false using the drop-down arrow.
6. Do the same thing for the enableFileSaving property.
7. Save your work, then run the application.

   Notice that you now have a context menu when you right-click the text area. Also notice that it does not contain menu items for Open and Save.

Note You can actually add any of the items on the context menu to your menu bar and toolbar by using DBTextDataBinder's public static Action classes, but you would have to provide the icons and write the code manually.

For an example of how to do this, see the TextPane sample in the JBuilder samples folder: <jbuilder>/samples/dbswing/TextPane.

See also

- The API documentation on the DBTextDataBinder component. To read it,
  a. Click the Source tab of TextEditFrame.java.
  b. Select dBTextDataBinder1 in the structure pane. It's inside the TextEditFrame component.

     The dBTextDataBinder1 declaration is highlighted in the editor.

c Put your cursor inside the class name `DBTextDataBinder`.

d Right-click and select Find Definition.

The `DBTextDataBinder` source file opens in the editor.

e Click the Doc tab to view the API documentation.

Close the "Text Editor" application before continuing to the next step.

Step 15: Showing filename and state in the window title bar

In this final step, we will add code that uses the title bar of the application to display the current filename, and to display an asterisk if the file is dirty.

We'll create a new method that will update the title bar, then call the method from places where the code changes either the current file name or the `dirty` flag. This new method will be `updateCaption()`.

1 Click the `jMenuFileExit_actionPerformed(ActionEvent e)` method in the structure pane. This moves the cursor to that event handling method and highlights it in the editor.

2 Place the cursor just *above* this method and insert the following `updateCaption()` method block:

```
// Update the title bar of the application to show the filename and its
dirty state.
void updateCaption() {
  String caption;

  if (currFileName == null) {
     // synthesize the "Untitled" name if no name yet.
     caption = "Untitled";
  }
  else {
    caption = currFileName;
  }

  // add a "* " in the caption if the file is dirty.
  if (dirty) {
    caption = "* " + caption;
  }
  caption = "Text Editor - " + caption;
  this.setTitle(caption);
}
```

Now, put the method call `updateCaption();` from each of the places the `dirty` flag actually changes or whenever you change the `currFileName`. The new method call location is indicated in a comment line in each code block below.

1. Put the call `updateCaption();` inside the `try` block of the `TextEditFrame()` constructor, as the next line immediately after the call to `jbInit()`. The `try` block will look like this:

```
//Construct the frame
public TextEditFrame() {
  enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  try  {
    jbInit();
    updateCaption();    // <---- HERE
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}
```

2. Put it as the last line in the `try` block of the `openFile()` method, which will look like this:

```
try {
  // Open a file of the given name.
  File file = new File(fileName);

  // Get the size of the opened file.
  int size = (int)file.length();

  // Set to zero a counter for counting the number of
  // characters that have been read from the file.
  int chars_read = 0;

  // Create an input reader based on the file, so we can read its data.
  // FileReader handles international character encoding conversions.
  FileReader in = new FileReader(file);

  // Create a character array of the size of the file,
  // to use as a data buffer, into which we will read
  // the text data.
  char[] data = new char[size];

  // Read all available characters into the buffer.
  while(in.ready()) {
    // Increment the count for each character read,
    // and accumulate them in the data buffer.
    chars_read += in.read(data, chars_read, size - chars_read);
  }
  in.close();
```

```
  // Create a temporary string containing the data,
  // and set the string into the JTextArea.
  jTextArea1.setText(new String(data, 0, chars_read));

  // Cache the currently opened filename for use at save time...
  this.currFileName = fileName;
  // ...and mark the edit session as being clean
  this.dirty = false;

  // Display the name of the opened directory+file in the statusBar.
  statusBar.setText("Opened "+fileName);
  updateCaption();    // <---- HERE
}
catch (IOException e)
{
   statusBar.setText("Error opening "+fileName);
}
```

3 Put it right before `return true;` in the `try` block of the `saveFile()` method:

```
try
{
  // Open a file of the current name.
  File file = new File (currFileName);

  // Create an output writer that will write to that file.
  // FileWriter handles international characters encoding conversions.
  FileWriter out = new FileWriter(file);
  String text = jTextArea1.getText();
  out.write(text);
  out.close();
  this.dirty = false;

  // Display the name of the saved directory+file in the statusBar.
  statusBar.setText("Saved to " + currFileName);
  updateCaption();    // <---- HERE
  return true;
}
catch (IOException e) {
  statusBar.setText("Error saving "+currFileName);
}
return false;
```

4 Make it the last line of code in the `if` block of the File | New menu handler `jMenuItem1_actionPerformed()`:

```
void jMenuItem1_actionPerformed(ActionEvent e) {
  // Handle the File|New menu item.
  if (okToAbandon()) {
     // clears the text of the TextArea
     jTextArea1.setText("");
    // clear the current filename and set the file as clean:
    currFileName = null;
    dirty = false;
    updateCaption();    // <---- HERE
  }
}
```

5 When the `dirty` flag is first set in a clean file due to user typing. This is done in each of the `document1` event handlers. The event handlers should be changed to read:

```
void document1_changedUpdate(DocumentEvent e) {
  if (!dirty) {
    dirty = true;
    updateCaption();    // <---- HERE
  }
}

void document1_insertUpdate(DocumentEvent e) {
  if (!dirty) {
    dirty = true;
    updateCaption();    // <---- HERE
  }
}

void document1_removeUpdate(DocumentEvent e) {
  if (!dirty) {
    dirty = true;
    updateCaption();    // <---- HERE
  }
}
```

6 Run your application and watch the title bar as you perform the following operations:

- Change the file name using File | SaveAs.
- Type in the text area, making the file dirty. Notice the * appear in the title bar as soon as the file has been touched.
- Save the file, making it clean.
- SE and Enterprise users, try out these actions using the context menu.

Congratulations! You have used JBuilder's visual design tools to create a functional text editor written entirely in Java. Users of JBuilder Personal edition, you have completed your tutorial. Please feel free to compare your code to the code in the sample, `<jbuilder>/samples/SimpleTextEditor`.

Step 16: Deploying the Text Editor application to a JAR file

This step is for JBuilder SE and Enterprise only.

Now that you've created the "Text Editor" application, you can deploy all the files to a Java Archive File (JAR) using JBuilder's Archive Builder.

Note If you haven't yet completed Steps 1 through 15 of this tutorial, you can still complete this step of the tutorial using the Text Editor sample project in the `samples/TextEditor/` directory of your JBuilder installation. To do this, you need to convert the paths specified in the tutorial to point to `samples/TextEditor/` and its subdirectories.

Overview

Deployment is an advanced subject which takes some study and experience to understand. JBuilder's Archive Builder reduces this complexity and helps you create an archive file that meets your deployment requirements.

This step of the tutorial gives you instructions for deploying the "Text Editor" application explicitly. It is not intended to be a comprehensive example of all the situations you'll run across when deploying Java programs. Each application or applet you deploy has its own unique set of deployment issues, so it's difficult to generalize. Links are provided throughout this step for further information on deployment, including Sun's Java™ Tutorial.

The first step in deploying any program is to identify which project and library contents will be included in the archive. This will help you determine what classes, dependencies and resources to include. Including all classes, resources and dependencies in your archive creates a large archive file. However, the advantage is that you don't need to provide your end-user with other files as the archive contains everything needed to run the program. If you exclude some or all classes, resources or dependencies, you'll need to provide them to your end-user separately.

The Archive Builder will not include the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment or Java Plug-in, or that you will be providing it in your installation.

JBuilder's Archive Builder creates an archive node in the project pane, allowing easy access to the archive file. At any time during development, you can make the archive file, rebuild it, or reset its properties. You can also view the contents of the archive and the contents of the manifest file.

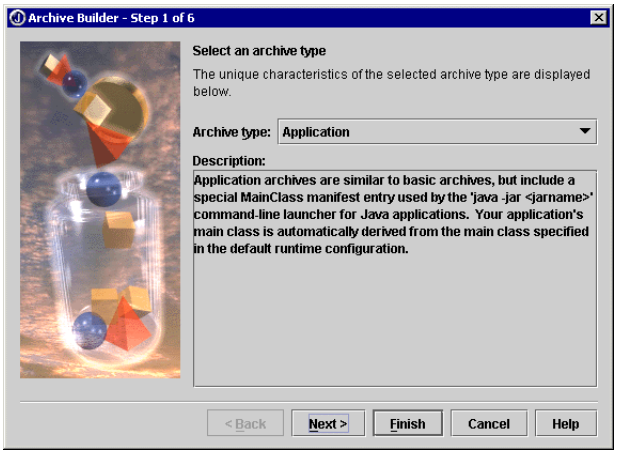
Running the Archive Builder

To run the Archive Builder wizard and create the archive node and file for the Text Editor tutorial,

1. Save all files in the project and compile it.
2. Choose Wizards | Archive Builder.

   Step 1 of the Archive Builder appears.
3. Choose Application for the Archive Type.

Step 1 of the wizard should look like this:



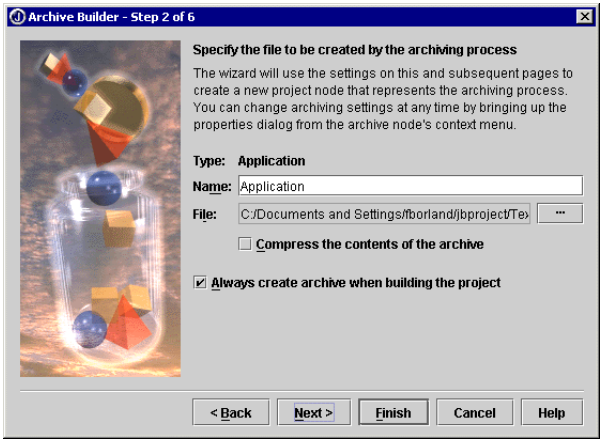
4 Click Next to go to Step 2 of the wizard.

5 Change the name of the archive to `Text Editor Application JAR` in the Name field. This is the name of the archive node that will be displayed in the project pane.

6 Accept the default JAR file name and path: `<project path>/TextEditor.jar`.

7 Accept the remaining defaults on this page.

When you're done, Step 2 of the wizard should look like this:



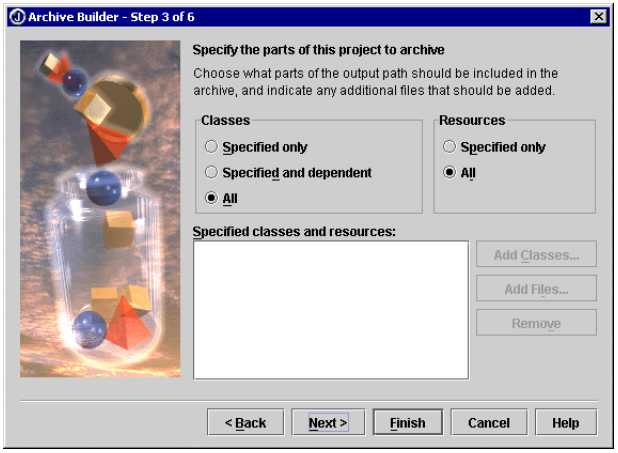
8 Click Next to go to Step 3 of the wizard, where you determine what project classes and resources are deployed.

The project classes and resources are those on your output path, defined on the Paths page of the Project Properties dialog box. Usually, this is set to the `classes` directory of your project. For this tutorial,

accept the default, so that the wizard includes all classes and resources on the output path.

Important Although this option is the safest and simplifies deployment, it makes the archive file larger. If for some reason your project includes unnecessary files on the output path, they are also included, making your deployed file very large. In this case, you might consider selecting the first or second option and manually adding classes and files with the Add Classes and Add Files buttons. Then test the deployed application to be sure you've included all the necessary files.

Step 3 of the wizard will look like this:



9 Click Next to go to Step 4 of the wizard.

In this step, you choose how library contents are included in your archive file. Usually libraries are not included in the archive file but are supplied as separate JAR files and included on the `CLASSPATH` at runtime. This is the easiest way to deploy and creates the smallest program JAR file. However, in this example, you'll include the libraries in the archive.

a Select dbSwing from the list.

b Choose the Include Required Classes And All Resources option from the Library Settings area.

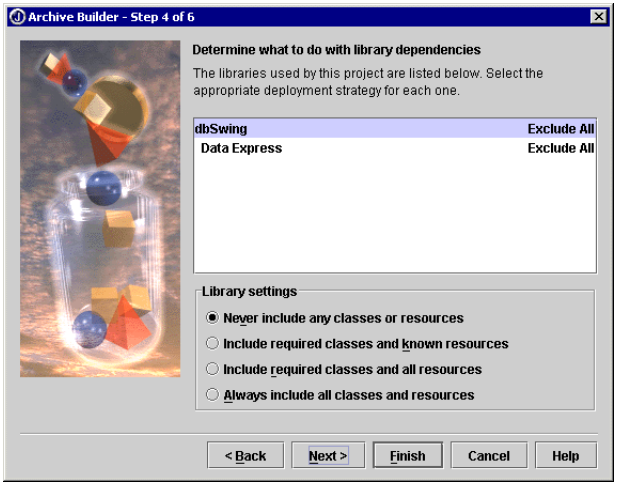
c Select DataExpress from the list and choose Include Required Classes And All Resources.

Even though you did not use the DataExpress library in this tutorial, some dbSwing classes depend on DataExpress classes. Therefore, they need to be included in the archive file.

Note that both libraries are deployed with Deps & Resources.

Caution The Archive Builder may not always find all the files. It's recommended that you test the deployed application, add any missing files, then redeploy.

Step 4 of the wizard should look like this:



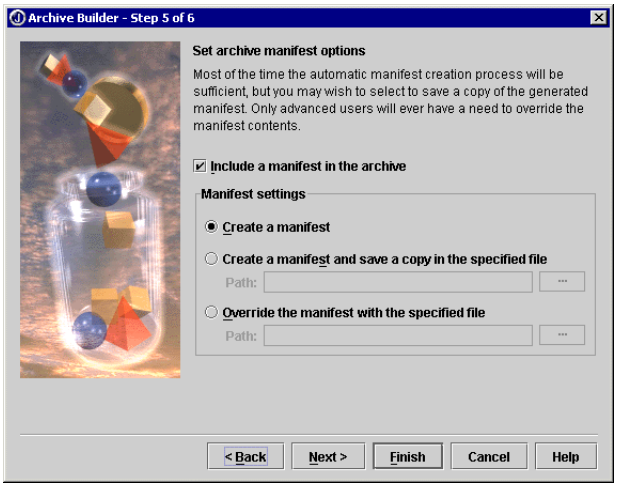
10 Click Next to go to Step 5, where you create the manifest file.

There can only be one manifest file in an archive, and it always has the path name `META-INF/MANIFEST.MF`.

11 Accept the default settings for Step 5 of the wizard. These have the following result:

- Automatically include the manifest file in the archive file.
- Automatically create the manifest file for you.

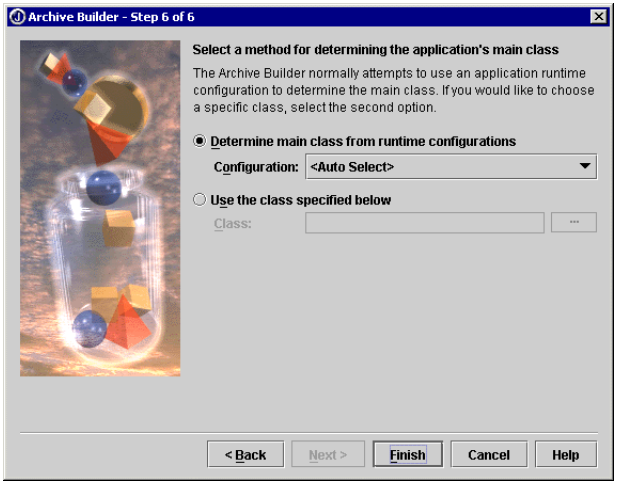
Step 5 of the wizard will look like this:



12 Click Next to go to Step 6, where you choose how the Archive Builder finds the main class.

For this tutorial, leave the default setting Determine Main Class From Runtime Configurations. This option uses the main class in the default runtime configuration specified on the Run page of the Project Properties dialog box.

Step 6 of the wizard will look like this:



13 Click Finish to create the archive node.

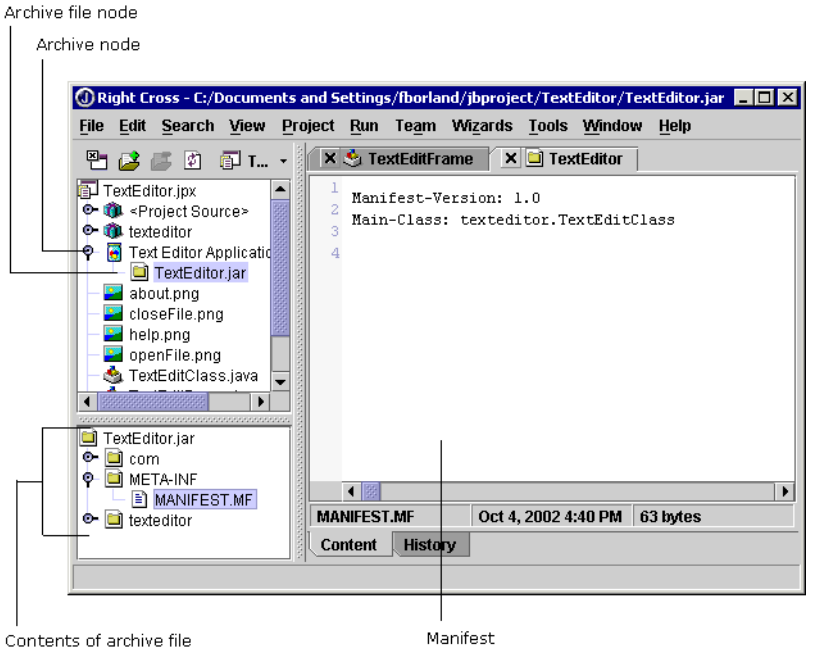
The archive node, `Text Editor Application JAR`, is now displayed in the project pane. You can right-click the archive node and make it, rebuild it, or change its properties.

14 Select Project | Make Project or right-click the archive node and choose Make to make the project and generate the JAR file.

15 Expand the archive node in the project pane to see the archive file.

16 Double-click the archive file, `TextEditor.jar`.

Its contents are displayed in the structure pane and the contents of the manifest file are displayed in the content pane. JBuilder should now look similar to this:



Notice the following two headers in the manifest file:

`Manifest-Version: 1.0`	Indicates that the manifest's entries take the form of "header:value" pairs and that it conforms to version 1.0 of the manifest specification.
`Main-Class: texteditor.TextEditClass`	Indicates that `TextEditClass.class` is the entry point for your application (the class containing the `public static void main(String[] args)` method, which runs the application.)

See also

- "Using the Archive Builder" in *Building Applications with JBuilder*.
- "Understanding the Manifest" at `http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html`.

Testing the deployed application from the command line

Before you run the application from the command line, you need to make sure your operating system's PATH environment variable points to the JDK jre/bin/ directory, the Java runtime environment. The JBuilder installation process guarantees that JBuilder knows where to find the JDK class files. However, once you leave the JBuilder environment, your system needs to know where the class files for the Java runtime are installed. How you set the PATH environment variable depends on which operating system you are using.

To run the Text Edit tutorial from the command line,

1. Switch to your command-line window and change to the TextEditor directory where the JAR file is located.
2. Check to see if Java is on your PATH by typing java at the command line. If it is, the Java usage and options will display. If it isn't on your PATH, set your PATH environment variable to the JDK's bin directory.
3. Enter the following command at the command line:

   ```
   java -jar TextEditor.jar
   ```

   where,

   - java is the Java tool that runs the jar file.
   - jar is the option that tells the Java VM that the file is an archive file.
   - TextEditor.jar is the name of the archive file.

   Since the manifest file provides the information in the Main-Class header about which class to run, you don't need to specify the class name at the end of this command. And, because all classes, resources, and dependencies are included in the archived JAR file, you don't need to specify a classpath or copy JBuilder libraries to this directory.

Note When you use the -jar option, the Java runtime ignores any explicit classpath settings. If you run this JAR file when you're not in the TextEditor directory, use the following Java command:

```
java -jar -classpath <full_path> <main_class_name>
```

The Java runtime looks in the JAR file for the startup class and the other classes used by the application. The Java VM uses three search paths to look for files: the bootstrap class path, the installed extensions, and the user class path.

If the application doesn't run, examine the errors generated in the command-line window. Make sure the jbuilder.lib folder is on your classpath. Make sure you are in the correct directory and there aren't any spelling errors in the command.

4 Test the application when it runs to be sure it's working correctly. Create, save, and open a file. Right-click in the text editor to see if the context menu is working. Also, the application could be running and still have errors. Check the command-line window for any error messages. Read the error messages, if any, to look for missing classes or packages.

See also

- "How Classes Are Found" at `http://java.sun.com/j2se/1.3/docs/tooldocs/findingclasses.html` to learn more about how Java searches paths.

Modifying the JAR file and retesting the application

If you have runtime errors, you need to add any missing classes to the JAR file using the Archive Builder. If you don't have errors, you can skip these steps.

1 Return to the Text Editor project in JBuilder.

2 Right-click the Text Editor Application JAR node in the project pane and choose Properties.

3 Select the appropriate tab and make any necessary changes.

4 Click OK to close the Properties dialog box.

5 Right-click the archive node and choose Make to rebuild the JAR file.

6 Repeat the testing procedure with the modified JAR file as described in "Testing the deployed application from the command line" on page 9-47, and test the application when it runs.

That's it!

As you can see, there is a lot of information to assimilate related to deployment. Deployment goes far beyond just creating an archive file. Not only do you have to make sure you provide all the necessary classes, resources, and libraries in your deployment set, you have to be concerned with other issues, such as learning about the java tool and the Jar tool. There are also differences between running JDK 1.1 and Java 2 applications.

Take the time to study the wealth of information available at the links to Sun's web site provided here, in other reputable online sources, and in the many excellent third-party books on the subject.

See also

- "Deploying Java programs" in *Building Applications with JBuilder*.
- Sun's web page on Basic Tools at `http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html#basic`.
- The Sun Tutorial trail on Jar files at `http://java.sun.com/docs/books/tutorial/jar/index.html`.

Chapter

10

Tutorial: Creating a UI with nested layouts

This tutorial steps you through creating a user interface for a hypothetical application like a simple word processor. It shows you how to create the UI with JBuilder's visual design tools, using nested panels and the simpler layout managers. It uses `BorderLayout`, `FlowLayout`, and `GridLayout`. Due to their complexity, `GridBagLayout` and `CardLayout` are discussed in detail elsewhere. For more information on layout managers, see Chapter 8, "Using layout managers," and Sun's "Creating a GUI with JFC/Swing" tutorial at `http://java.sun.com/docs/books/tutorial/uiswing/index.html`.

In this tutorial, you'll use the Swing `JPanel` for all panel components, because unlike the AWT `Panel`, it has a `border` property. You will use the `border` property to add borders to the panels so you can see their boundaries when you add components to them in the designer. Once your UI design is finished, you can remove the borders wherever you like. The `JPanel` component is located on the Swing Containers tab of the component palette at the top of JBuilder's UI designer. Throughout this tutorial, any reference to a panel implies `JPanel`.

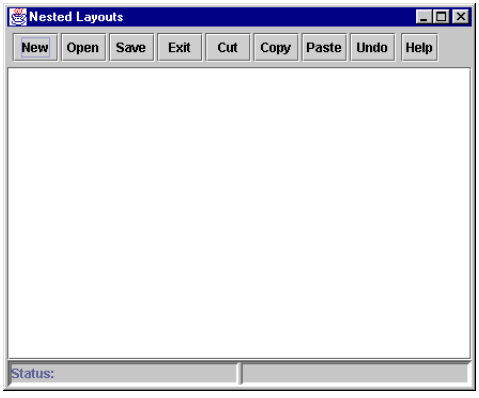
Also in this tutorial, you'll change some of the panel layout managers during the design phase to `XYLayout` (JBuilder SE or Enterprise) or `null` layout (JBuilder Personal). `XYLayout`, a JBuilder custom layout manager, places components in a container at specific x,y coordinates relative to the upper left corner of the container. Regardless of the type of display, the container will always retain the relative x,y positions of components. `null` layout means that no layout manager is assigned to the container. `null` layout (from Swing) is very similar to `XYLayout` in that you can put components in a container at specific x,y coordinates relative to the upper left corner of the container.

Using XYLayout's or null's absolute positioning makes designing your UI easier, because you can control component positioning and sizing. But the disadvantage is that they do not adjust well to differences in systems, and as a result, they are not portable layouts. Because XYLayout and null use absolute positioning, containers and components do not resize correctly when the user resizes the application window. Therefore, it's best not to leave a container in XYLayout or null for deployment due to the lack of portability. AWT layout managers, which don't use absolute positioning, make it easy to adjust your application to different system look and feels, various system font sizes, and to a container's changing size. Therefore, they are more portable than XYLayout and null. So, after the entire UI design is populated in this tutorial, you will change the layout managers of all the containers to the more portable AWT layouts.

For the best layout control in a UI design plus a design that is simpler and less deeply nested, it's best to use a combination of GridBagLayout and the nesting techniques demonstrated in this tutorial. For an in-depth tutorial on GridBagLayout, see Chapter 11, "GridBagLayout tutorial." If you are serious about doing Java UI development, it's important to take the time to learn how to use GridBagLayout. Once you understand it, you'll find it indispensable.

The application user interface you are about to design contains several panels that hold components, such as buttons, labels, and a text area. Because you are focusing on user interface design in this tutorial, the application you design is not fully functional. For example, if you click the Save button on the toolbar, nothing happens. Also, this tutorial is not the only way to design this user interface. For instance, you would normally use the JToolBar component when creating toolbars and GridBagLayout. In this tutorial, you use panels and buttons for the toolbar to demonstrate the use of nested panels and layouts. The buttons could be any type of component in your design.

Here is the UI you're going to design:



Note The screenshots in this tutorial use the Metal Look & Feel in JBuilder's integrated development environment and in the application's runtime environment.

Step 1: Creating the UI project

Before creating your application, you must create a project for the project and applications files. Once you've created the project, you'll use the Application wizard to generate the source files. Create the project using the Project wizard.

Using the Project wizard

To open the Project wizard,

1. Choose File | New Project to open the Project wizard.
2. Make the following changes in Step 1:
   - Name: `NestedLayouts`

   Note By default, JBuilder uses this project name to create the project's directory name and the package name for the classes.
   - Check the Generate Project Notes File option. When you check this option, the Project wizard creates an HTML file for project notes and adds it to the project.
   - Make sure the Add Project To Active Project Group option is unchecked. You don't want to add this project to any of your real work.
3. Accept all other defaults on Step 1.
4. Click Next to go to Step 2 of the Project wizard.
5. Accept the default paths in Step 2.
6. Click Next to continue to Step 3 of the Project wizard.
7. Fill out the class Javadoc fields. This information is saved in the project HTML file. It's also used for Javadoc comments if you choose the Generate Header Comments option in many of JBuilder's wizards, such as the Application and Class wizards.
8. Press Finish to create the project. A project file and a project HTML file are added to the project and appear in the project pane.

See also

- "Creating and managing projects" in *Building Applications with JBuilder*.

- "Managing paths" in *Building Applications with JBuilder*.
- "How JBuilder constructs paths" in the "Managing paths" chapter in *Building Applications with JBuilder* for more information on Step 2 of the Project wizard.
- "Where are my files?" in the "Managing paths" chapter in *Building Applications with JBuilder*.

Step 2: Generating the application source files

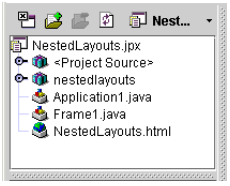
Now that you have a project, you can use the Application wizard to automatically generate your application files.

Using the Application wizard

To open the Application wizard,

1. Choose File | New to open the object gallery.
2. Choose the General tab, if it's not already the active tab.
3. Double-click the Application icon to open the Application wizard. Note the default package name which is extracted from the project name.
4. Click Finish. We won't use the additional files generated in Step 2 of the Application wizard, and we're going to use the default values in Step 1 of the wizard.

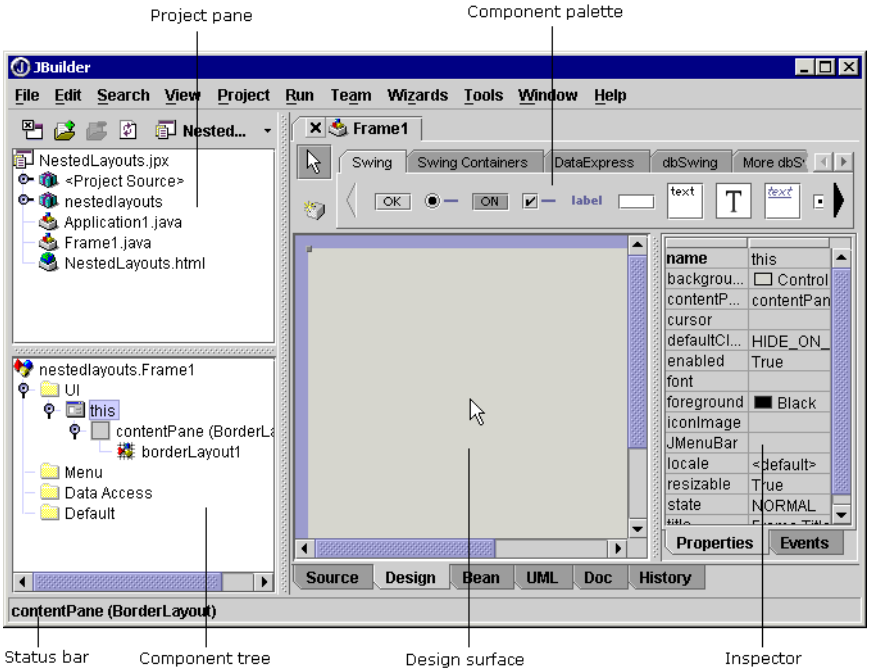
The project appears in the project pane of the AppBrowser with `Frame1.java` open in the content pane. You'll see two additional files in the project pane: `Application1.java`, which contains the `main()` method, and `Frame1.java`, the UI container.



Note In JBuilder SE and Enterprise editions, an automatic source package node also appears in the project pane if the Automatic Source Packages option is enabled on the General page of the Project Properties dialog box (Project | Project Properties).

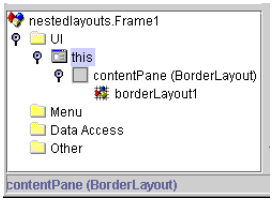
1. Click the Design tab at the bottom of `Frame1.java` in the content pane to open the visual designer in UI design mode.

Figure 10.1 UI designer



The structure pane now contains a component tree and the content pane contains the UI designer, with the Inspector on the right and the component palette above the design surface.

Notice the structure of your UI design in the component tree as created by the Application wizard. You have a frame called `this`, under which is the `contentPane (BorderLayout)` placed in the `UI` folder. It is under this folder and frame that JBuilder places the visual UI components as you add them to your design. You'll actually be designing the `contentPane` in the UI designer. Notice also that `this` is highlighted in the component tree and sizing nibs are on each corner of the frame in the designer. The frame is anchored at its top left corner. The properties for `this` are displayed in the Inspector to the right of the designer.

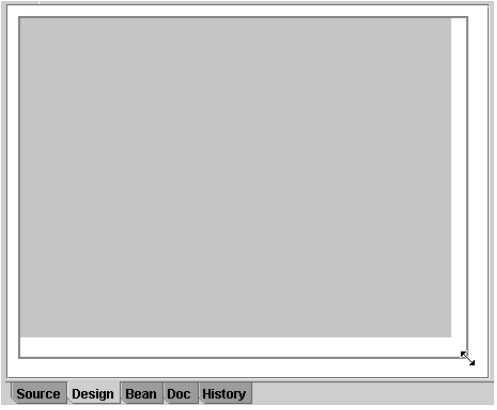


Tip The status bar below the structure pane displays information on any component that the mouse hovers over in the designer.

2. Look at the Properties page of the Inspector and find the value of the `title` property, which is "Frame Title".
3. Triple-click `Frame Title` to highlight it, type `Nested Layouts` in its place, then press *Enter*. The new title displays in the frame's title bar at runtime.

   Now, make the surface area in the UI designer larger, so you have more room to work.
4. Click the bottom right corner nib with the mouse and drag it diagonally away from the center of the frame to enlarge it in the designer. You are actually resizing the main container frame for your UI, the component instance `Frame1` of the `JFrame` class.

Tip If you can't see the nib on the bottom right corner, maximize the AppBrowser before you drag the frame to a larger size. If you want even more space, hide the project and structure panes by selecting View | Toggle Hide All. Note, however, that closing the curtain hides the component tree that is in the structure pane, so you'll have to toggle the curtain every time you want to see the tree. You can also make the designer larger by dragging its borders.



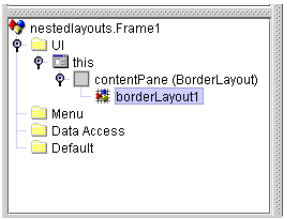
The actual screen size of a UI container at runtime is not necessarily determined by the size you make it in the UI designer. Initial runtime screen size is determined by the container's layout manager.

You can manually change the size of the UI frame by resizing it at runtime.

Step 3: Changing contentPane's layout

A Java UI 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. Layout managers provide your UI design with such advantages as portability across platforms, dynamic resizing of components at runtime, and ease of translation with strings of different sizes.

A layout manager automatically arranges the components in a container based on the layout manager's layout rules and property settings, the layout `constraints` associated with each component, common component properties (`preferredSize`, `minimumSize`, `maximumSize`, `alignmentX`, and `alignmentY`), and the size of the container. By default, `contentPane`'s layout manager is `BorderLayout`. You can see the layout manager by clicking the icon to the left of `contentPane` in the component tree and expanding the tree.



`BorderLayout` is best used in UI design when placing five or fewer components where one center component requires the most space. `BorderLayout` arranges components in five locations: Center, North, South, East, and West, with Center being the largest.

If you leave the `contentPane` in `BorderLayout`. you might accidentally drag a panel with the mouse to another area of the `BorderLayout` frame, causing the panel to shrink in height or width and move to one of the edges of the frame. If this happens after you have the UI design populated with lots of components, it might be difficult for you to immediately determine what the problem is.

To prevent this from happening, change the `contentPane` to `XYLayout` or `null`, layouts that allow more control over the positioning of components.

1. Select `contentPane` in the component tree.
2. Select its `layout` property on the Properties tab of the Inspector.
3. Click the drop-down arrow and select `XYLayout` or `null` from the list.

Starting with `null` or `XYLayout` makes prototyping your design easier. Later, after adding components to the container, you can switch to an appropriate portable layout for your design. It's best not to leave a container in `null` or `XYLayout`, because they use absolute positioning and,

therefore, components do not adjust well when you resize the parent containers. These layouts also do not adjust well to differences in users and systems and as a result are not portable layouts. It's best to use these layouts in the design phase and then convert everything to the portable layouts at the end of the design phase. You'll change the layout manager back to `BorderLayout` at the end of the tutorial, after all the other panels are converted to their ultimate layouts.

Step 4: Adding the main panels

Now, begin adding panels to your UI design. When you're done, the design should look something like this:

1. Select `contentPane` in the component tree.
2. Click the Swing Containers tab on the component palette at the top of the designer and click the `JPanel` icon.

Tip Move the mouse over a component on the component palette to see its name in a tool tip.

3. Draw in the first panel by clicking and dragging diagonally from the top left corner of the designer to the right side of the `contentPane`, creating a panel that fills the top fourth of the design. This panel will be the container for two toolbars at the top of the application.

   Notice that a new component called `jPanel1` is added to the `UI` folder in the component tree under the `contentPane`. You can see sizing nibs around the edges of the panel showing its size and location. Click the expand icon to see `jPanel`'s layout manager, `FlowLayout`. `FlowLayout` is used when placing a few components in a row. Because you'll be placing two toolbar panels in a row in this top panel, leave the layout as `FlowLayout`.

4 *Shift+click* the `JPanel` icon on the component palette and draw two more panels in the designer. For the moment, don't worry about making the layout perfect. You'll fine-tune the layout later. Notice that the second and third panels are also using `FlowLayout`. You'll change these later before adding components. Check the component tree to see that all three panels are nested inside the `contentPane`.

Tip If you *Shift+click* the `JPanel` component on the component palette, you can add multiple panels without clicking the `JPanel` icon each time. This is particularly useful when adding multiple, identical components to a layout. When you're done adding the panels, click the Selection arrow to the left of the palette to deselect the `JPanel` component.

5 Click the Selection arrow on the component palette to deselect the multiple selection feature. Each of these three panels you just added will contain other components. The top panel will have two panels, each containing a toolbar with buttons. The middle panel will contain a scroll pane and a text area. And the bottom panel will contain a status bar with two labels. To make each panel more distinguishable in the designer, change the border property to RaisedBevel. You'll use a shortcut by selecting all the panels and changing the border property for all of the panels at the same time.

6 Select `jPanel1`, `jPanel2`, and `jPanel3` using *Shift+click* to select all of them. You can select them in the designer or in the component tree.

7 Choose RaisedBevel from the `border` property drop-down list on the Properties page of the Inspector.

While it isn't necessary to use a border on the panels during design, it does make the design work easier when you are nesting multiple panels and components. This is because as soon as you select a component on the palette, then click the panel, the panel's sizing nibs disappear and you can't see if you are still inside the panel when you drag the new component to its desired size.

For the purpose of demonstration, the images in this tutorial also show the panels in contrasting shades of gray to make it easier to differentiate them. Changing the background color of the panels is another way you could make them visible in the designer if you don't want to use borders.

Adjust the layout using some of the designer's menu selections. For example, you can make these three panels the same width horizontally.

1 Cancel the multiple selection in the component tree by selecting another component.

2 Select the top panel in the designer and use the sizing nibs to adjust its width.

3 Hold down the *Shift* key, and click the middle and bottom panels. Now all three panels are selected.

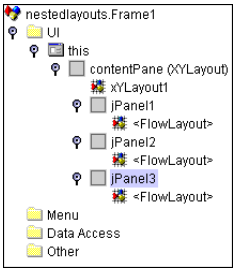
4 Right-click with the mouse over one of the selected panels and choose Same Size Horizontal. The middle and bottom panels will snap to the same width as the top panel. Then select Align Left and Even Space Vertical from the designer's context menu.

5 Click any component not selected or click the component tree to cancel the multiple selection.

Tip The first component selected is matched, so be careful which one you select first.

To drag a panel to another position, select the panel and move the mouse over the center black nib in the panel. You'll see a four-headed arrow appear. Click and drag the component to the new location. Be careful not to drag it out of its container and into another container. If you make a mistake, just choose Edit | Undo and the design will return to its previous state.

Notice the structure of your UI design now in the component tree:



Now, rename the three panels you just added, so they have more meaningful names.

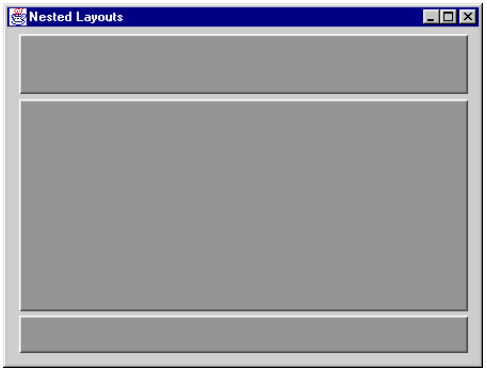
1 Select each panel in the component tree. Right-click the panel and select Rename from the menu. Give each panel the appropriate name:

- top panel: `top`
- middle panel: `middle`
- bottom panel: `statusbar`

At this point, it would be a good idea to save the entire project and run the application.

2 Choose File | Save All from the main menu.

3 Choose Run | Run Project or click the Run icon on the toolbar. Your application should look something like this:



You may need to modify the size of the `this` frame or the position of your panels after running the application. Before enlarging any panels, select `this` and make it larger. Also, when you run the application, resize the application window. Notice that the panels do not resize with the window. This is why you do not want to leave your design in `XYLayout` or `null`. At the end of the tutorial, when you change `contentPane`'s layout back to `BorderLayout` the panels will resize correctly with the window.

4 Exit the application.

5 Right-click the Application1 tab on the message pane and select Remove All Tabs to close the message pane.

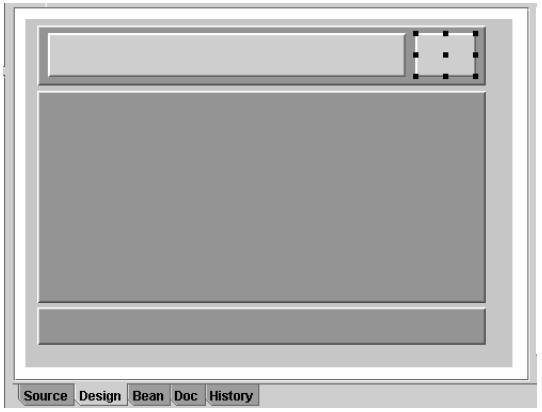
Step 5: Creating toolbars

Before adding panels to the top panel, which will contain two toolbars, you'll change the layout manager of the top panel to `XYLayout` or `null`, so you'll have complete control over the position of the panels you are adding. Then, you'll add two panels and rename them to more meaningful names. The first panel will hold the left toolbar and the second panel will contain the right toolbar.

1 Select the top panel.

2 Change the layout manager from `<default layout>` (`FlowLayout`) to `XYLayout` or `null`.

3 Draw two panels on the top panel using the same method as in step 1 and matching the look of the image below. Don't worry about their size and position yet. We'll be fine-tuning them later.



4 Select each of the panels in the component tree and rename each panel on the top panel as follows:

- top left panel: `left_toolbar`
- top right panel: `right_toolbar`

5 Change the borders on both panels to RaisedBevel.

Now, let's adjust the horizontal height of the toolbar panels and align them to the top.

6 Hold down the *Ctrl* key and select `left_toolbar` and `right_toolbar` in the component tree.

7 Right-click over one of these selected panels in the UI designer and choose Align Top.

8 Right-click again with the two panels still selected and choose Same Size Vertical.

9 Save your work.

Step 6: Adding toolbar buttons

Before adding buttons to your toolbars, you must change the two toolbar panel layout managers to `GridLayout`, so the buttons you add will be the same size. `GridLayout` is used to place components of identical sizes in a grid of rows and columns.

1 Select `left_toolbar` and `right_toolbar` and change their `layout` properties to `GridLayout` in the Inspector. Now, let's add the buttons to the two toolbar panels and label them.

2 Choose the Swing tab of the component palette.

3 *Shift*+click the `JButton` component and click eight times on the `left_toolbar` panel in the component tree. The first button completely fills the panel and, as each button is added, `GridLayout` makes them the same size.

4 Click the Selection arrow on the component palette to deselect the `JButton` component selection.

5 Select each button in turn, starting with the far left button, and change its `text` property in the Inspector as follows:

- `New`
- `Open`
- `Save`
- `Exit`
- `Cut`
- `Copy`
- `Paste`
- `Undo`

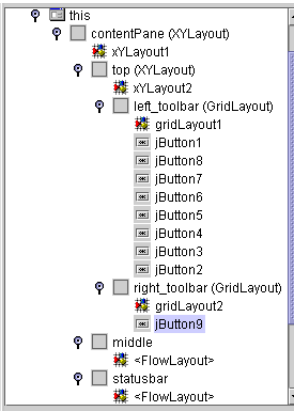
Important You may not be able to see the text on the buttons yet because the margins need to be adjusted. You'll do that to all the buttons after you finish adding them.

Note If your buttons are not in the correct order, right-click a button and choose Move To First or Move To Last from the designer's context menu.

6 Put `jButton9` on the `right_toolbar` panel and change its text to `Help`.

The important thing now is that the buttons are fully nested inside their panels. To be sure they are all embedded properly, check the component tree to see if each button is indented under the correct panel in the tree outline. If any buttons did not get nested inside their panels, you'll see them in the component tree at the same level of indentation as the panels.

The component tree looks something like this, although your buttons may be in a different order:



Next, make the button text readable by reducing the margins.

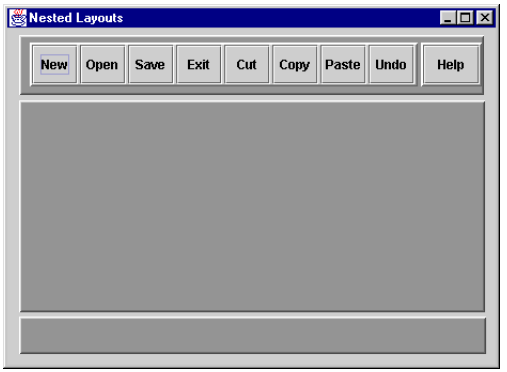
1. Select all the buttons on both panels in the designer or the component tree. You can use the *Shift* key to select consecutive components in combination with the *Ctrl* key to select the Help button on the right toolbar in the component tree.
2. Change the `margin` property to `2,2,2,2`.

Note If you can't see all the text on the buttons, make them wider by first selecting `this` in the component tree and making it wider first, then `top` next, working inward to widen the rest of the panels.

Finally, put a little space between the buttons on the `left_toolbar` panel. You do this by changing the horizontal gap on the layout manager itself. Notice that the first item under each container in the component tree is its layout manager.

1. Select the `GridLayout` object for `left_toolbar`.
2. Change the `hgap` property in the Inspector to 2 and press *Enter*.
3. Save your work again and run the application. Notice the space added between the buttons.

Your design should now look something like this:



Optional

If the buttons don't look the same at runtime as they did in the designer, it is probably because the look and feel set in JBuilder is different from that on your system. Changing JBuilder's look and feel (Tools | IDE Options) changes it for the JBuilder IDE but doesn't change the runtime look and feel. The screenshots in this tutorial use Metal for both JBuilder and runtime. To set the look and feel you want at runtime, open `Application1.java` and change the following line:

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

to one of the following:

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

4 Save your work and close the application and the message pane.

Step 7: Adding components to the middle panel

Now let's work on the middle panel that will contain a scroll pane and a text editing area. First, you must change the layout manager to `BorderLayout`. This layout is a good choice when placing five or fewer components and especially when you want a component to completely fill the layout. `BorderLayout` has five areas: Center, North, South, East, and West with the largest area given to the Center. In this example, you want the scroll pane and the text area to completely fill the `middle` panel with a constraint of Center.

1 Select the `middle` panel in the component tree and change its `layout` manager to `BorderLayout`.

2 Choose the Swing Containers tab on the component palette.

3 Select the `JScrollPane` component and drop onto the Designer. `JScrollPane` is used to display a component, such as a text area, that is

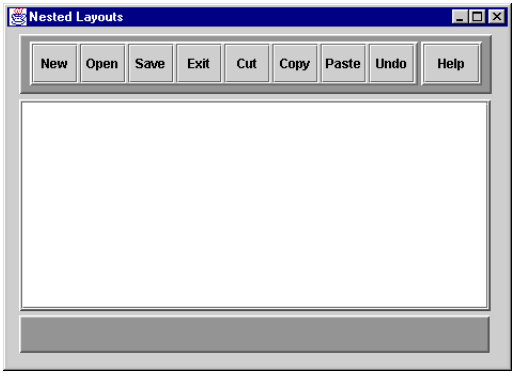
too large to display or that changes dynamically. JScrollPane should fill the middle panel. If it doesn't change, JScrollPane's constraints property to Center in the Inspector.

4 Choose the Swing tab on the component palette and select the JTextArea component. Drop it on JScrollPane in the designer or the component tree. It should completely fill JScrollPane.

Note Swing components with a text property have a default text value entered into the text property. You can remove this by highlighting the text displayed in the text property value and pressing *Delete*, then *Enter*.

5 Save your work again and run the application to see how the UI looks.

This is what your UI should look like now:



6 Exit the application and close the message pane.

Step 8: Creating a status bar

Now, work on the last panel of your UI design, the status bar. Although this area, like the others, won't be fully functional, you'll make it look like a UI status bar. Your status bar should look something like this when you're done:

Create the statusbar panel as follows:

1 Change the statusbar panel's layout to GridLayout. Now, when you add the labels, they will be the same size, just like the buttons on the toolbar.

2 Add two Swing JLabel components to the statusbar panel as shown. Make sure they are contained by the statusbar panel.

3 Select the two labels and change their border properties to LoweredBevel to achieve a three dimensional look seen in many status bars.

4 Select the `GridLayout` object for `statusbar` in the component tree and change the `hgap` property to 2 to widen the raised area between the status bars. This step is purely a matter of taste and not a necessity.

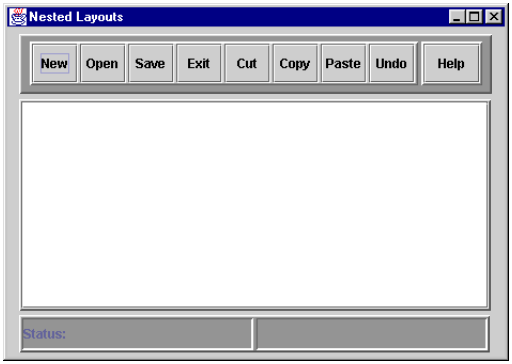
Important In JBuilder, you cannot edit the `layout` properties for a `<default layout>`. If you want to modify the properties for a container's layout manager, you must specify an explicit layout manager. Then its properties are accessible in the Inspector.

5 Select the left label and change the default value in the left label's `text` property to `Status:`.

6 Select the right label and delete the default value in the right label's `text` property and press *Enter*.

7 Save your work and run the application.

Your application should look something like this:



8 Exit the application and close the message pane.

At this point, you have completed the first phase of the UI design work, adding all the components to the containers, starting with the larger outside containers and working down to the smallest components inside these containers.

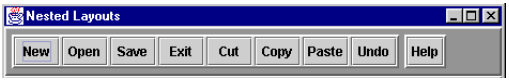
Step 9: Converting to portable layouts

In this step, you'll begin working from the inside out, converting panels to more portable layouts. Remember, you don't want to leave anything in `XYLayout` or `null` due to their absolute positioning of components and lack of portability.

Now, change the layout for the top panel that contains the toolbars and align the toolbars to the left.

1. Select the `top` panel and change the `layout` property in the Inspector to `FlowLayout`. The two panels in the top panel, `left_toolbar` and `right_toolbar`, now flow from left to right.
2. Select the top panel's `FlowLayout` object in the component tree and change the `alignment` property to Left. Usually toolbars are aligned left in UI design.
3. Save and run your application and notice that the toolbars are left-aligned in the UI.

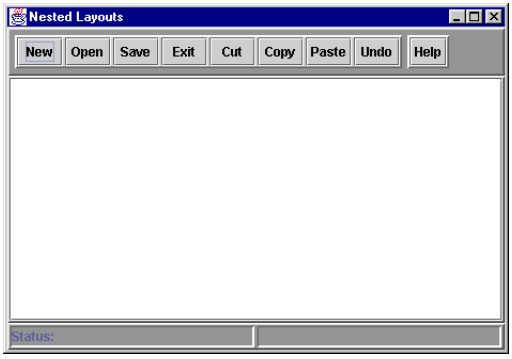
   Now your top panel should look similar to this:



4. Exit the application and return to the designer.
5. Select the top panel in the designer and make it narrower than the buttons to see what `FlowLayout` does when the panel is narrower. Notice that the Help button on the `right_toolbar` panel moves to the second row. This is `FlowLayout` behavior. Adjust the panel until both toolbars are back on the top row.

   Finally, change the layout for `contentPane` to `BorderLayout`.
6. Select `contentPane` in the component tree and change `XYLayout` to `BorderLayout`. It should assign the `top` panel to the North, `middle` panel to the Center, and `statusbar` to the South. If this doesn't happen, select each panel and correct its `constraints` property in the Inspector.
7. Save your work and run the application.

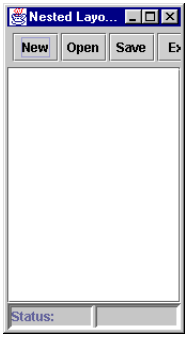
Your UI components should all be in their correct places now. If your design is too large or too small, return to the UI designer, select `this` in the component tree and resize the frame.



Try resizing the application window now that `XYLayout` has been replaced with other more portable layouts. Note the behavior of `BorderLayout`: when you enlarge the window, the center area gets as much space as possible

and the other areas, in this case North and South, expand enough to fill the remaining areas. North and South can only expand horizontally to fill the space and can't grow vertically in height. This becomes a problem when you make the window narrower.

Now, make the window very narrow and observe the top panel containing the toolbars. The top panel can't resize due to the number of buttons filling it, so the buttons are hidden.



This occurs because the buttons are at their minimum size for displaying the text and `BorderLayout` North can only stretch horizontally not vertically. Therefore, the buttons can't wrap. From this example, you can see that `BorderLayout` is not the best choice for more complicated layouts with toolbars. It's best used when you only have several components that need to fill Center, North, South, East, and West. On the other hand, `GridBagLayout`, in combination with other layout managers, is ideal for more complicated UI designs and well worth learning.

Step 10: Completing your layout

Everything is finished except for a little cleanup and polish. If you did change the color of any panels, now is your opportunity to change them back to their original gray.

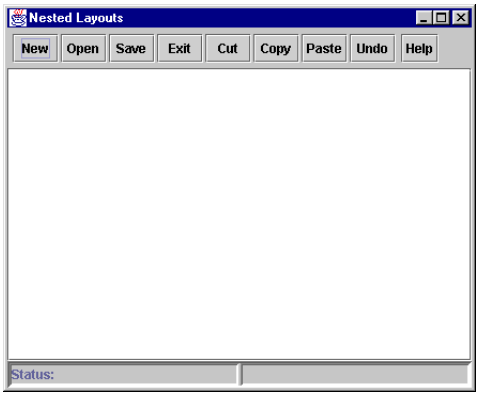
1. Select all the panels in your design using multiple selection in the component tree.
2. Double-click the `background` property in the Inspector and select light gray (RGB value 192, 192, 192). Don't worry about its appearance in the designer at this point. The bevels create a lot of white space. You might also want to select all the buttons and make their backgrounds the same as the panels.

   Next, eliminate any unwanted borders.
3. Select all the panels in your design for which you want to remove the borders and change the `border` property to `<none>`.

This is purely your choice and may be influenced by which look and feel you decide to use. You can even customize a border by clicking the ellipsis button to the right of the `border` property to open the Border Property Editor.

In this tutorial all the borders have been removed except for the LoweredBevel border on the status bar labels.

Now, the UI is complete except for any other finishing touches you might want to add. This is approximately how your final design should look:



And that's all there is to it! While it seems slow in the learning phase, once you become familiar with the different layouts, you'll be able to plan and implement layouts more quickly.

It is easy to see that using multiple levels of panels with only the easier layouts can actually be pretty tedious and complicated. A better alternative is to take the time to learn `GridBagLayout`. In the end, all your UI designs will be much simpler and better controlled. You will still use nested layouts, but only one or two levels deep. `GridBagLayout` will control the rest of the layout behavior.

For an in-depth tutorial on `GridBagLayout`, see Chapter 11, "GridBagLayout tutorial."

To learn how to write code that responds to user events in your application, see the online tutorial called "Building a Java text editor."

Chapter

11

GridBagLayout tutorial

Introduction

This is an in depth tutorial that explains `GridBagLayout`, and demonstrates how to create a `GridBagLayout` UI container using the JBuilder visual design tools. The goal of this tutorial is to give you a thorough understanding of how `GridBagLayout` works in JBuilder and to show you how to simplify `GridBagLayout` design. While the information here is aimed at working with JBuilder, much of it also applies to working with `GridBagLayout` in general.

The images in this tutorial were captured on the Windows platform. This does not, however, affect the validity of the tutorial for other platforms, as the basic functionality of JBuilder and `GridBagLayout` is the same on all platforms.

Important This tutorial uses `XYLayout` for prototyping the UI. If you use JBuilder Personal, substitute `null` layout wherever `XYLayout` is mentioned.

The "GridBagLayout tutorial" is divided into three sections:

- "Part 1: About GridBagLayout" on page 11-2

  Part 1 of the tutorial explains conceptually what the `GridBagLayout` manager and the `GridBagConstraints` objects are. It gives you a detailed description of each constraint and explains how to set the constraint in the JBuilder visual designer. This part also explains why `GridBagLayout` can be so complicated and shows you how you can simplify `GridBagLayout` design by using the designer.

- "Part 2: Creating a GridBagLayout in JBuilder" on page 11-16

  Part 2 walks you through the steps of creating a typical dialog box using `GridBagLayout`. It demonstrates how to plan the UI before you start

and gives you examples of the differences in behavior of the container with different layout choices.

- "Part 3: Tips and techniques" on page 11-39

  Part 3 is a collection of various tips and techniques for working with `GridBagLayout` in JBuilder. In this section, each constraint's behavior is examined separately, with examples that show you what to expect when modifying it in the designer. A code sample generated by JBuilder as a result of creating the UI example in Part 2 is included. This part also explains how to change existing `GridBagLayout` code to be visually designable in JBuilder.

  Part 3 also includes:

  "GridBagConstraints" on page 11-62

  An overview of the `GridBagConstraints` and their values.

  "Examples of weight constraints" on page 11-67

  Illustrated examples of weight constraints applied in different ways.

Part 1: About GridBagLayout

Overview of GridBagLayout

XYLayout is a feature of JBuilder SE and Enterprise. If you use JBuilder Personal, substitute null layout wherever XYLayout is specified.

`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 special features to the visual design tools that make `GridBagLayout` much easier to design and control, such as a GridBagConstraints Editor, a visual grid, drag-and-drop editing, and a special context menu for components in a `GridBagLayout` container.

There are two approaches you can take to designing `GridBagLayout` in the visual 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` or `null` layout, then convert it to `GridBagLayout` when you have all the components arranged and sized the way you want. This method can speed up your design work substantially and is the one which is the focus of this tutorial.

Whichever method you use, you should take advantage of using nested panels to group the components. Use 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.

For more information on using nested panels, see "Using nested panels and layouts" on page 8-51.

What is GridBagLayout?

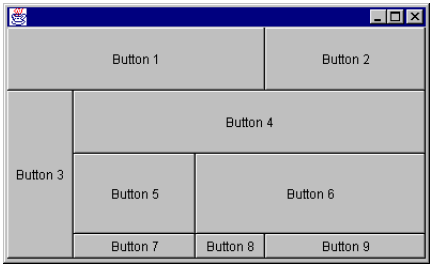
In Java, you create a user interface by adding components to a Container object, such as `Frame` or a `Panel`, and using a layout manager to control the size and placement of the objects within the container. By default, every container object has a layout manager object that controls its layout.

`GridBagLayout` is an extremely flexible and powerful layout manager that implements the interface LayoutManager2 and knows where and how to layout objects based on object GridBagConstraints. It places components horizontally and vertically on a dynamic rectangular grid, but provides more control in the size and location of the components than `GridLayout` (in which the grid cells are of equal size, filled with one component each).

Unlike `GridLayout` the components in `GridBagLayout` do not have to be the same size and they can span multiple cells. Also, the columns and rows in the grid do not have to be the same width or height.

`GridBagLayout` controls the placement of its components based on the values in each component's `GridBagConstraints` object, the component's minimum size, and the container's preferred size.

Figure 11.1 Example GridBagLayout



The key advantage to `GridBagLayout`, as shown in this example, is the ability of the components to grow or shrink reliably. This feature provides greater application portability between platforms, computer resolutions, and localization of products where string lengths change.

In the example above, some of the buttons occupy only one cell of the grid (one row, one column), while others span multiple cells or rows and columns. You can see the exact number of cells each component occupies in the designer when you display the grid for a `GridBagLayout` container. The difference between a cell and the area each component occupies is explained in the next topic "What is the component's display area?".

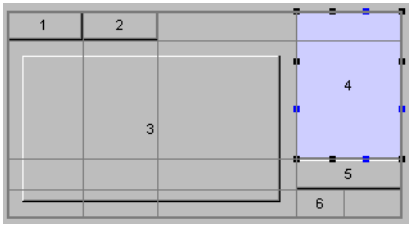
What is the component's display area?

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 horizontally (column) and two cells vertically (rows). Therefore, its display area consists of two cells.

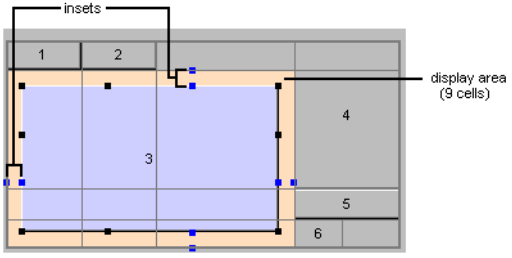
Figure 11.2 Display area — one horizontal cell, two vertical cells



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 barrier between the edges of the display area and the component.

Figure 11.3 Display area — three horizontal cells, three vertical cells



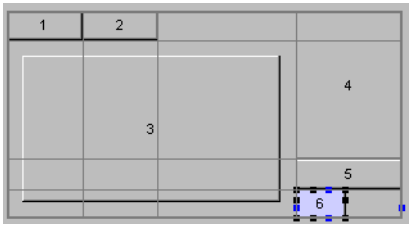
Even though this component has both horizontal and vertical `fill` constraints, since it also has `insets` on all four sides of the component (represented by the double blue nibs on each side of the display area),

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, plus leaving 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.

Figure 11.4 Component smaller that its display area



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, "What are `GridBagConstraints`", and in "Part 3: Tips and techniques" on page 11-39.

What are 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.

`GridBagConstraints` give you control over the following:

- The absolute or relative position of each component.
- The absolute or relative size of each component.
- The number of cells each component spans.
- How the unused space in a component's display area gets filled.
- The amount of internal and external padding for each component.

- How much weight is assigned to each component to control which components utilize any extra available space. This controls the component's behavior when resizing the container, or displaying the UI on different platforms.

`GridBagLayout` components have the following constraints:

- `anchor`
- `fill`
- `gridx, gridy`
- `gridwidth, gridheight`
- `insets`
- `ipadx, ipady`
- `weightx, weighty`

See also

- java.awt.GridBagConstraints.html at `http://java.sun.com/j2se/1.3/docs/api/java/awt/GridBagConstraints.html`
- java.awt.GridBagLayout.html at `http://java.sun.com/j2se/1.3/docs/api/java/awt/GridBagLayout.html`

Why is GridBagLayout so complicated?

When you first start modifying the constraints in a `GridBagLayout`, you can often have unexpected results that can seem dramatic and difficult to understand. The difficult thing about learning how to assign constraints to components in the `GridBagLayout` container is knowing what effect the change to one component has on the other components in the grid. The constraint behavior for one component depends on the other components in the container and their constraints. For example, if you decide to remove `weight` values from a component, the position of the other components in the grid might change relative to this.

The two examples below show the effect of changing the `weighty` constraint value of Area 4 from 1.0 to 0.0. Notice how the row collapses, and the bottom row expands.

Figure 11.5 Area 4 weighty constraint value is 1.0:

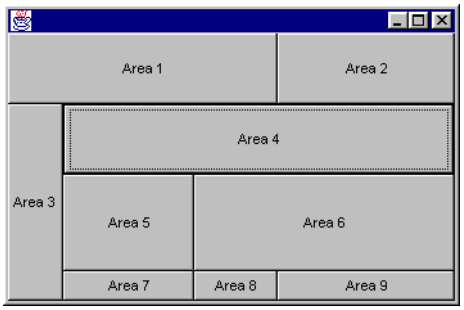
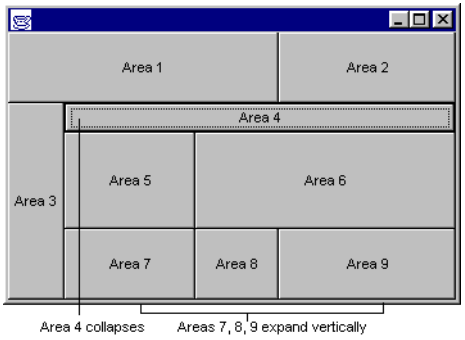


Figure 11.6 Area 4 weighty constraint value changed to 0.0:



JBuilder shortens the `GridBagLayout` learning curve time by allowing you to see the effects of changes immediately in a visual design surface.

Why use GridBagLayout?

`GridBagLayout` gives you complete control over how components behave and how they are displayed when the container is resized or viewed on different platforms. This ensures that your distributed application will look and behave properly on any supported platform.

Most books and tutorials tend to skip an in-depth discussion of `GridBagLayout`, and many recommend avoiding using it altogether. It is possible to accomplish much of your UI design work by using a combination of other layouts.

If you have tried to use `GridBagLayout` before now, you've discovered that it is very complex and initially difficult to work with. Getting the design to work exactly the way you want involves tedious trial and error, modifying the constraints in the code then compiling and running to see if it works. Until you fully understand the behavior of the individual constraints and the effect their modifications will have on the design, GridBagLayout can be extremely frustrating.

As with any complex subject, the easiest way to work with it is to simplify it

Simplifying GridBagLayout

JBuilder gives you a much simpler way to do `GridBagLayout` design work, making it possible for even the beginning Java programmer to use. By using a combination of the visual designer, JBuilder's `XYLayout` or `null` layout, and JBuilder's excellent layout conversion ability, all of the initial layout and design code is generated automatically, taking most of the guesswork out and leaving only minor adjustments to do.

Whether you are new to Java or are an advanced programmer, the suggestions below can significantly improve your experience with `GridBagLayout` and speed up your UI design work:

- Sketch your design on paper first
- Use nested panels and layouts
- Use the JBuilder visual designer
- Prototype your UI in XYLayout

Sketch your design on paper first

Always start your `GridBagLayout` design on paper. Take the time to sketch the final design and decide where it would be best to include nested panels with other layouts. Nested panels are essential for simplifying the design and giving you absolute control over the placement of the components.

For example, if you want a toolbar in your `GridBagLayout` design, use a nested `GridLayout` panel to contain the buttons, rather than placing the buttons directly into the `GridBagLayout` container. Try to arrange your design so you have a minimum number of panels and components for the `GridBagLayout` container to control.

At first, it may not be obvious how important it is to plan ahead on paper. But, if you start prototyping without a plan, you soon discover how much time you would have saved if you had logically thought it through before beginning. Eventually, your knowledge and skill with the various layout managers will become advanced enough that you may be able to skip this step and just begin your prototype in the designer. But, in the beginning, it is well worth the time and extra effort it takes to plan it out. See "Step 1: Design the layout structure" on page 11-17.

Difficulties adding components after conversion to GridBagLayout

While planning the layout before you start is good advice for any UI design, it is especially important for `GridBagLayout`. When you add

components to an existing GridBagLayout container, or move existing ones, unpredictable results may surprise and overwhelm you. If you can anticipate the final layout requirements before you start, you can minimize the amount of final adjustments needed after you convert the layout from XYLayout to GridBagLayout. See "Prototype your UI in XYLayout" on page 11-14.

The following example demonstrates what can happen when you add a component to a panel after converting it from XYLayout to GridBagLayout. This example uses the same layout design you're going to create in Part 2 of the tutorial, and demonstrates adding the first of three buttons at the bottom of the design. However, in this instance, none of the components in the panel are grouped into nested panels, and the conversion to GridBagLayout was done before trying to add the buttons at the bottom. You can easily see how hard it is to control the location of just the first button!

Notice in the first two images below the difficulty GridBagLayout has figuring out where to put the new button. Even though the button is drawn in the middle of the bottom row, GridBagLayout snaps it into the first column.

Figure 11.7 Drawing the new button in the middle

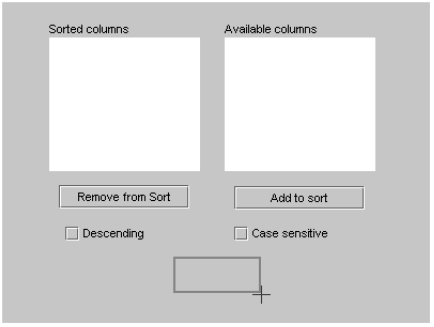


Figure 11.8 GridBagLayout snaps the button to column 1 when it is dropped

In the next two images, the button is dragged back to the middle. `GridBagLayout` changes the number of columns in the layout as it tries to accommodate the new button location.

Notice that when the cursor is directly in over the middle line between the two columns, `GridBagLayout` snaps the button to the top. If you place the cursor on either side of the middle, the button will be snapped into an existing column, rather than creating a new one.

Figure 11.9 Dragging the new button back to the middle

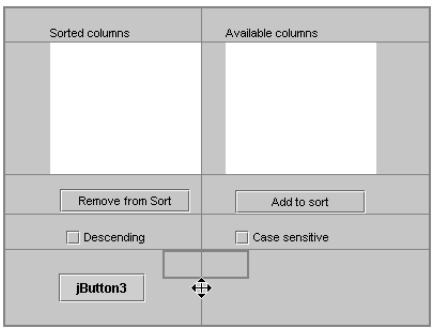


Figure 11.10 GridBagLayout creates a new center column for the button

Unfortunately, since the components above are not placed into two separate panels, this alters the constraints (and size) of the other components because the components on the right now span two columns.

It's obvious to see from these examples that adding the button while the UI was still in `XYLayout` instead would have enabled us to place it exactly where and how we wanted without affecting the preferred size and placement of the other components.

Note Actually, as you'll see later, adding a panel across the bottom to contain the buttons gives you greater control over their placement in `GridBagLayout`.

See also

- "Prototype your UI in XYLayout" on page 11-14.

Use nested panels and layouts

Most UI designs in Java use more than one type of layout to achieve the desired results. You can often get the best control by nesting multiple panels with different layouts in the main UI container. You can also nest panels within other panels to gain more control over the placement of components. By creating a composite design and using the appropriate layout manager for each panel, you can group and arrange components in a way that is both functional and portable.

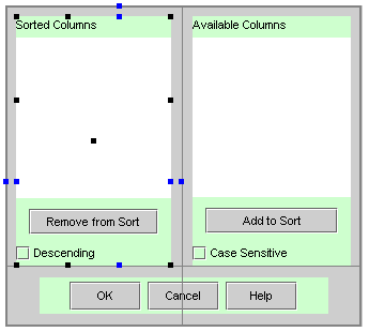
While `GridBagLayout` can accommodate a complex grid, it behaves more successfully and predictably if you organize your components into smaller panels, nested inside the `GridBagLayout` container. These nested panels can use other layouts, including `GridBagLayout`, and can contain additional panels of components if necessary. This method has several 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 toolbars.
- It minimizes the actual number of components being controlled by `GridBagLayout`, greatly simplifying the design.
- It reduces the chances of unexpected behavior when modifying constraints.
- It minimizes the need for further modifications after conversion to `GridBagLayout`.

Note It's best to group components into nested panels if grouping can make it easier to keep the grid divided into fewer evenly placed cells. The fewer components you have in a `GridBagLayout` container, the easier it is to control placement of the components.

For example, in the UI design used for this tutorial, you can fit most of the components into two columns, except for the three buttons at the bottom.

Figure 11.11 UI design for GridBagLayout tutorial



Placing three buttons at the bottom of the panel increases the total number of columns, making the alignment of the other components trickier. Also, getting those three buttons to stay the same size in the middle of the dialog box when the container is resized is harder to achieve if they are separate components in the larger `GridBagLayout` panel.

If you choose to leave the buttons in the larger `GridBagLayout` panel, when you really stretch the container horizontally, the buttons at the bottom get further and further apart, rather than staying together in the center of the panel.

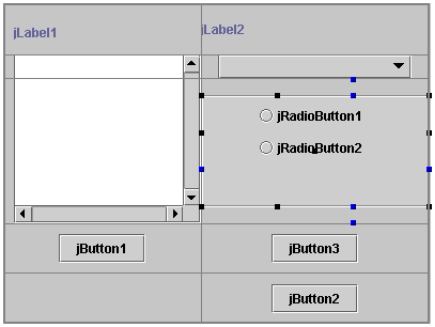
If, instead, you group the three buttons into one `GridLayout` panel, that panel can span two columns of the `GridBagLayout`. This makes it possible to have a total of two columns. This is much simpler for `GridBagLayout` to manage. Also, the placement of the buttons in the dialog box behave predictably when the container is resized, remaining together in the center.

Use the JBuilder visual designer

JBuilder's visual design tools make the challenging, time-consuming and risky work of using GridBagLayout much simpler, faster, and safer:

- By using the designer to design your UI, you can do all the initial design work in `XYLayout` which lets you control exact placement and size of the components. When you finish the design work, you then switch the layout to `GridBagLayout`. JBuilder does all the work of calculating the constraint values of the components in the layout and automatically generates the code for you, greatly simplifying and speeding up the entire design process.
- The designer displays a visual grid to aid your `GridBagLayout` design work. This grid appears whenever you click on a component in a `GridBagLayout`, and lets you see individual cells and the relationship between the components and their cells. This grid can be turned on or off and hides when you click on a component in a different type of layout.

Figure 11.12 Designer with the grid turned on:



- You can use drag-and-drop editing on the design surface to visually modify a component's constraints. Each component in the `GridBagLayout` container displays a set of nibs for adjusting the size, location, insets, or padding of the component.

  With the grid visible, this allows you to see exactly what is happening to the entire grid and its components when you drag a component or one of its nibs, as shown in the image above. The values are also immediately updated in the source code and in the GridBagConstraints Editor.

  To turn the grid on in the designer, right-click a component in the `GridBagLayout` container and choose Show Grid.

Note Use of these sizing nibs is discussed in the "Tips and techniques" section under the individual constraints that use them.

- You can assign or modify all constraint values in the Constraints property editor if you prefer, which is accessible from the context menu on the design surface or from the Inspector. The GridBagConstraints Editor remains open while you adjust and apply constraints for more than one `GridBagLayout` component, as long as you just click on `GridBagLayout` components. This gives you the ability to experiment with minor adjustments and see the results immediately.

  You can also modify constraint values directly in the source code because changes between the designer and the editor are always synchronized. Again, the results take effect immediately in the designer.

- JBuilder provides multiple levels of undo, making it easier and less risky to try out modifications. You can return to a previous state if unexpected things happen or if you don't like the change. In the beginning stages of learning to design `GridBagLayout`, this inevitably happens.

- For each object you add to a GridBagLayout container using the designer, JBuilder creates a GridBagConstraints object that has a constructor which takes all eleven of its properties:

```
public GridBagConstraints(int gridx,
                          int gridy,
                          int gridwidth,
                          int gridheight,
                          double weightx,
                          double weighty,
                          int anchor,
                          int fill,
                          Insets insets,
                          int ipadx,
                           int ipady)
```

  For example, if you add a button to a GridBagLayout panel called jPanel1, the following code is generated by JBuilder:

```
jPanel1.add(jButton1, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
  GridBagConstraints.CENTER, GridBagConstraints.NONE,
  new Insets(0, 0, 0, 0), 0, 0));
```

Prototype your UI in XYLayout

The main advantage to prototyping your UI design in JBuilder's XYLayout is that this layout keeps components at the exact pixel location and size you create them. You also have numerous alignment options available on the component's context menu for aligning multiple components: left, right, center, top, bottom, middle, same size horizontally or vertically, and equal spacing horizontally or vertically.

Using XYLayout, you can lay out your components exactly the way you want them to be, then convert the container to GridBagLayout, letting JBuilder calculate the grid cells and constraint values automatically. JBuilder does a good job of this conversion, but in many cases, you may want to fine-tune a few constraints to get the exact behavior you want. This is usually because of two factors:

- The conversion process may apply weight and fill constraints you don't want on particular components, giving you undesirable behavior.
- The number of cells JBuilder decides are necessary is usually more than you would guess. If you are trying to center something that spans multiple cells, like a toolbar for example, you might need to adjust the number of columns or rows the component spans (gridwidth or gridheight).

Even so, the bulk of the work of coding GridBagLayout has already been done for you, greatly speeding up the entire process. As you become more familiar with how constraints affect component behavior, and with how JBuilder does the conversion from XYLayout to GridBagLayout, you can better

anticipate what nested panels are necessary to make the design behave well.

Below are the basic work flow steps for using this design strategy:

1. Create the container that will ultimately be `GridBagLayout`. This can be the main UI container, or it can be a panel inside the UI container.
2. Change its layout to `XYLayout`, if necessary. The easiest way is to change the Layout property in the Inspector.
3. Add all the components to the container while it is still in `XYLayout`. Use nested panels to minimize the actual number of components being ultimately controlled by the `GridBagLayout`.
4. Get as close as possible to the finished layout design so the conversion to `GridBagLayout` is more successful. Take advantage of `XYLayout`'s alignment options on the context menu to fine-tune the placement, size, and alignment of the components.
5. When the UI is basically finished, convert the main container to `GridBagLayout`.

Important

Generally, when you use nested panels in a layout, you would convert the inner panels to their intended layout first, working outward to convert each level of containers, then converting the main container last. However, the strategy is different for `GridBagLayout`.

When converting a container from `XYLayout` to `GridBagLayout`, you should leave the inner panels in `XYLayout` until you have converted the outer container to `GridBagLayout`. This is because during the conversion process, JBuilder determines the number of columns and rows to create in the grid based on the preferred width and height of the components at the time of conversion. The conversion process honors the preferred width and height of the `XYLayout` panel. It determines the number of columns to create and the insets needed based on those dimensions.

For example, if you are trying, as in our example UI, to center a `GridLayout` toolbar panel across the bottom of the `GridBagLayout` container, converting that panel of buttons to `GridLayout` first shrinks the panel to fit the buttons. Depending on the width of the components in the rest of the `GridBagLayout` container, the conversion might not make the `GridLayout` panel span all the columns in the `GridBagLayout`.

By leaving the panel extended across the entire width of the container in `XYLayout` during the conversion to `GridBagLayout`, JBuilder knows it needs to center the panel and span it across all the columns in the container. The toolbar panel is well behaved inside the `GridBagLayout` container after you convert it to `GridLayout`.

6. Convert the inner panels to their intended layouts.

7 Make minor adjustments to constraints if needed to perfect your design. This mainly involves changing insets (for example matching left and right insets for components you want centered in the cell), and making sure the fill and weight constraints were applied the way you want.

   For tips on fine-tuning your design after you have converted to `GridBagLayout`, see "Part 3: Tips and techniques" on page 11-39.

8 Save and run your program. Resize the frame in different ways to check for any unwanted behavior. If necessary, make additional adjustments until you are satisfied with the results.

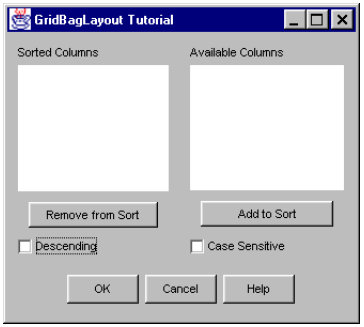
Part 2: Creating a GridBagLayout in JBuilder

XYLayout is a feature of JBuilder SE and Enterprise. If you use JBuilder Personal, substitute null layout wherever XYLayout is specified.

About the design

This part of the `GridBagLayout` tutorial takes you through each step of designing a `GridBagLayout` container in JBuilder. You will create the following typical dialog box that has several controls on it and a group of three buttons centered at the bottom.

Figure 11.13 GridBagLayout tutorial UI



The reason we decided upon this particular design is precisely because of the complications introduced by adding the odd number of buttons at the bottom. Also, this is a situation frequently encountered.

As you work through this part of the tutorial, please keep in mind that due to individual differences in drawing and arranging the components, your design may not exactly look like, nor behave like ours. But, it should be similar enough to allow you to achieve the desired results.

- Steps 1-3 of this tutorial involve creating the layout in the designer.
- Steps 4-6 walk you through converting it to `GridBagLayout`.
- Step 7 shows you how to adjust the individual constraints to fine tune your design and the reasoning behind these modifications.

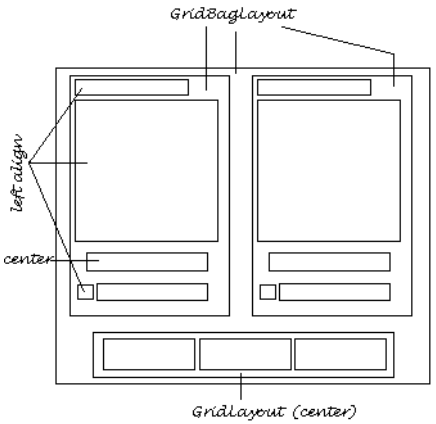
Take your time, and don't be afraid to experiment. The designer makes it easy to try out different things and immediately see the effects. Just save your `Frame1.java` file before you try anything so you can Undo back to that point after you're done.

Step 1: Design the layout structure

The first step in designing your UI is to sketch the design on paper to plan the container structure and layouts. See "Sketch your design on paper first" on page 11-8.

We have already done this step below.

Figure 11.14 Sketch of proposed design



In our sketch, we grouped the components into three panels for `GridBagLayout` to control: two equal sized panels to hold the components in the main part of the UI and one panel across the bottom for three buttons. We did this for two reasons:

- The components in the upper part fit conveniently into two columns. Grouping them into two panels means that the conversion to `GridBagLayout` creates only two columns with the bottom panel spanning the two columns and centering easily in the `GridBagLayout` panel.
- This design succeeds in trimming our total number of `GridBagLayout` components down to three, making the layout much simpler to control.

The images below demonstrate how JBuilder would handle the conversion using this arrangement. Notice that the grid displays only two columns and two rows in the designer. (The background color of the panels has been changed in this example to make it more obvious how JBuilder decided where the divisions should be.)

Figure 11.15 Conversion to GridBagLayout by designer

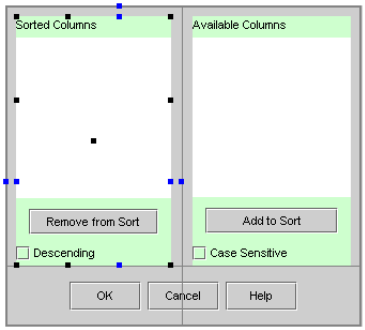


Figure 11.16 Runtime, before resizing

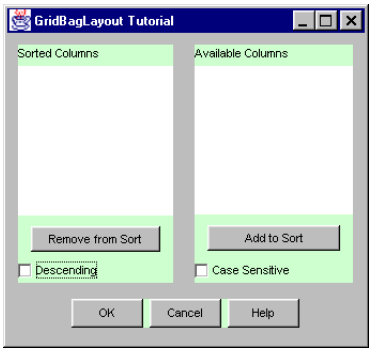
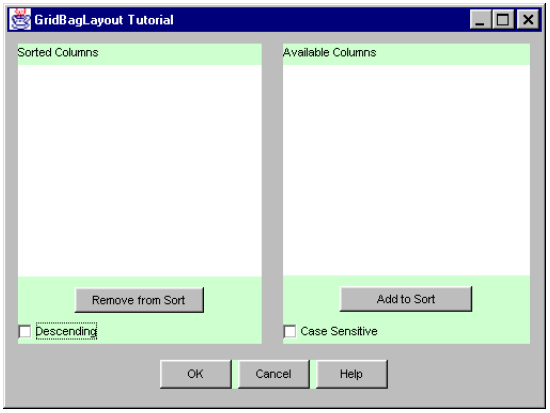


Figure 11.17 Runtime, after resizing

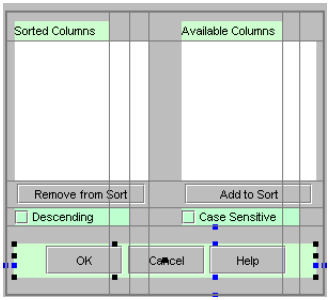


As shown in the next example, you could also use one panel for the three buttons at the bottom and let the `GridBagLayout` control all the other components separately, rather than nesting them inside panels. This can work, as long as you are very careful to make all the upper components the same width on each side. However, `GridBagLayout` creates more

columns based on where each component ends, making a much more complicated design that is more difficult to control.

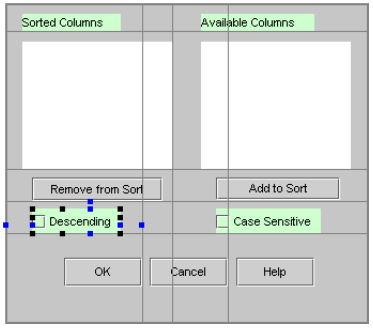
Notice that the conversion created six columns this time. (Again, the background color has been changed on the transparent components so you can see where they end, illustrating how the columns and rows were calculated.)

Figure 11.18 Conversion results without nested panels in upper columns



Finally, if you don't use any inner panels to group components, `GridBagLayout`'s job is much harder. In addition to creating even more cells, it makes a determination as to how many of these cells each component uses for a display area and which components get weight constraints. The more components in the design, the greater the potential for misinterpretation of your original intentions.

Figure 11.19 Conversion results without any nested inner panels



The images below show how a `GridBagLayout` panel with no inner panels behaves when it is resized at runtime:

Figure 11.20 GridBagLayout, no inner panels, before resizing

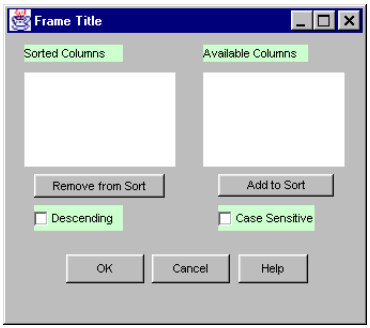
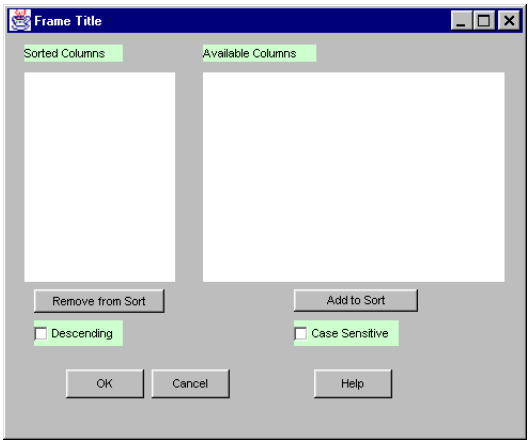


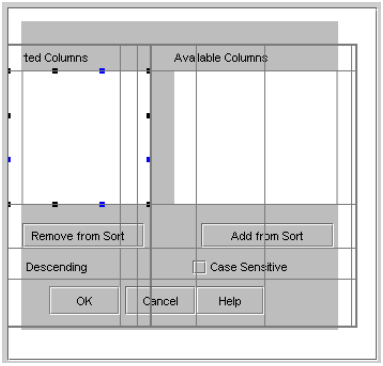
Figure 11.21 GridBagLayout, no inner panels, after resizing



As you can see, the components don't do what we want. Without grouping them into panels, it's practically impossible to control their placement and size during resizing.

Also, without any panels, if the components are not aligned very carefully and evenly, you could potentially end up with a grid that looks this:

Figure 11.22 Possible results if no inner panels, and components not carefully aligned



When your grid becomes larger than the `Frame`, it can be an indication that you have too many disparate objects for `GridBagLayout` to control, and the decisions it makes as to weight constraints, `gridwidth`, `gridheight` and `anchor`, cause the design to require a larger grid than the container can hold. When this happens, the display area of the objects are off the edge of the `Frame`.

If this does happen to you during `GridBagLayout` conversion, you may not want to waste time trying to correct it. It could take hours of frustration to clean it up by modifying constraints, and you still might not succeed satisfactorily.

Just do an Undo (*Ctrl+Z*) to reverse the conversion, returning to `XYLayout`. Readjust the design in `XYLayout`, then try converting it again.

For a cleaner conversion to `GridBagLayout`, try these changes while in `XYLayout`:

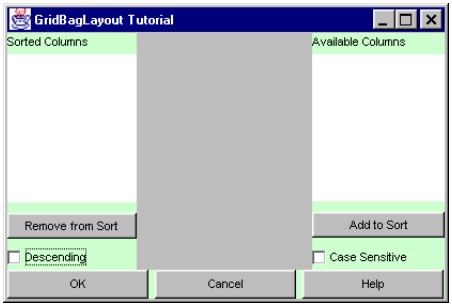
- Group more of the components into nested panels if possible.
- Make your design more symmetrical. Match up the beginning and ending of as many of the components as possible to minimize the number of cells required in the grid.

If you do decide to go forward when it is in this state, try enlarging the frame wide enough to display all the cells (especially the columns) so you can see how many there actually are in the design. That way you can modify how many cells each component is supposed to span, reducing the total number of cells in the grid. Then you can correct the `gridx` and `gridy` locations for each display area, as well as where the components are anchored in that display area. Also, you may want to try removing all weight constraints until the other constraints are fixed.

To avoid these difficulties, use nested panels wisely in your design.

It might occur to you that if you use four panels (three inside one `GridBagLayout` panel), it would work just as well to use `BorderLayout` for the main panel instead of `GridBagLayout`. The image below demonstrates how differently `BorderLayout` handles the components from `GridBagLayout`.

Figure 11.23 Results if using BorderLayout instead of GridBagLayout



Step 2: Create a project for this tutorial

JBuilder uses projects to organize associated files into folders.

To start a new project,

1. Choose File | New Project to open the Project wizard.
2. Modify the path and project name if you like, then click Finish.
3. Choose File | New to access the object gallery and select the General tab. Click on the Application icon on the General tab in the object gallery to open the Application wizard.
4. Select the Application icon and either double-click it or press *Enter*.
5. Accept all defaults and click Finish.
6. Save the project: choose File | Save Project.

Step 3: Add the components to the containers

Let's proceed with creating the UI design that uses three nested panels inside a main `GridBagLayout` panel to group the components.

Tip Since we will sometimes be working with more than one component, let's review how to handle multiple components in the designer.

- If you have any trouble determining where you are on the design surface, read the component's name in the status bar. The status bar tells you exactly which component your arrow hovers over on the design surface.

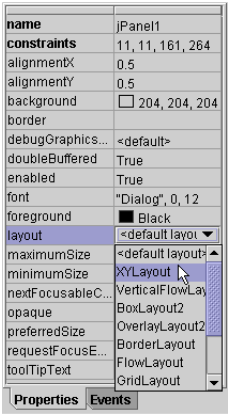
- If you have trouble selecting a component on the design surface, you can always select it in the component tree instead.
- You can select multiple components in either the component tree or on the design surface by holding down the *Ctrl* key as you click each component.
- When modifying the alignment for multiple components, the *first* component selected is the one to which the others match.
- To release a multiple selection, click on any unselected component in the designer or in the component tree.
- To display the right-click context menu for a panel containing multiple components, select the panel, place your cursor over the middle nib where a four-sided arrow cursor appears, then right-click. If you have difficulty selecting the panel in the designer, select it in the component tree, then move the cursor over the middle nib in the designer.

Figure 11.24 Select middle nib

Add the main panel to the UI frame

1. Select the `Frame` file in your project (`Frame1.java`) and click the Design tab to open the UI designer. The `this` component is the parent container of the UI. Its default layout, which you won't change here, is `BorderLayout`.
2. Click the Swing Containers tab on the component palette and select a `JPanel` component. Click in the center of the frame to add this panel. This places the panel (`jPanel1`) in the center and fills the entire frame.
3. Select `jPanel1` in the component tree or on the design surface. Select the Layout property in the Inspector and change the layout to `XYLayout`.

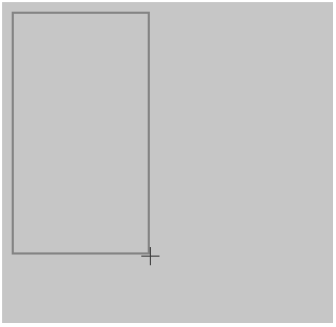
Figure 11.25 Change layout in Inspector



Create the left panel and add its components

1 Add a `jPanel` component from the Swing Containers tab to the upper left area of `jPanel1`. Stretch it to fit almost halfway across and about two thirds down, leaving some margin between it and the left edge of `jPanel1` as shown below. This is `jPanel2`.

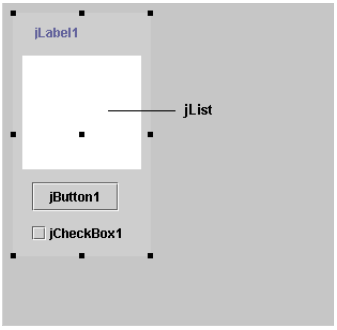
Figure 11.26 GridBagLayout tutorial, jPanel2



2 Change the layout for `jPanel2` to `XYLayout`. You can change the background color temporarily if you want to make it easier to see.

3 Add the following components from the Swing tab, starting in the upper left corner of `jPanel2`: `jLabel`, `jList`, `jButton`, and `jCheckbox`. You may need to stretch the `jList` component to enlarge it after adding it to the panel.

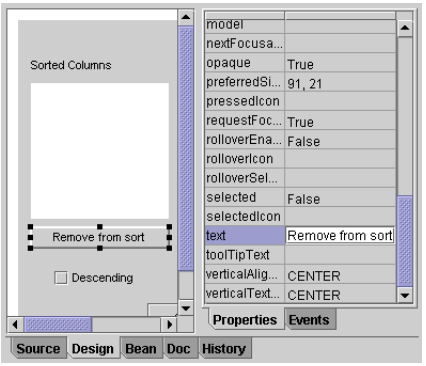
Figure 11.27 GridBagLayout tutorial, jPanel2 components



4 Change the `text` property in the Inspector for these components as follows:

`jLabel1` = "Sorted Columns"
`jButton1` = "Remove from Sort"
`jCheckbox1` = "Descending"

Figure 11.28 Text property



5 Match the font for `jLabel1`, `jButton1`, and `jCheckbox1`:

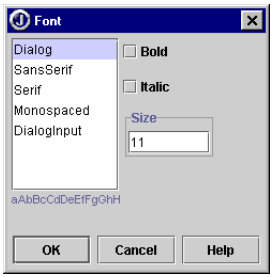
a Hold down the *Ctrl* key, then select `jLabel1`, `jButton1`, and `jCheckbox1` in the component tree.

b Click on the `font` property in the Inspector.

c Click the ellipsis button to bring up the Font dialog.

d Change the font from 12 pts to 11 pts if it's not already 11 pts, then click OK.

Figure 11.29 Font dialog



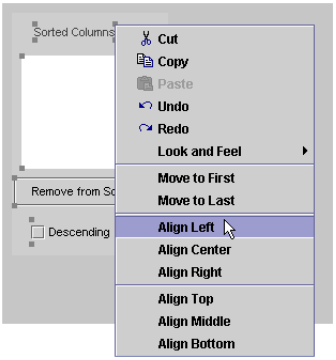
Even up the component sizes and alignment

1 Align the components using the `XYLayout` context menu. Hold down the *Ctrl* key and click on `jLabel1`, `jList1`, `jButton1`, and `jCheckbox1` on the design surface.

Tip When your cursor hovers over a component on the design surface, the name of the component appears in the status bar.

2 With the cursor still over one of the selected components, right-click on the design surface to bring up the context menu and choose Align Left.

Figure 11.30 Align left



3 Now make the `jLabel1`, `jButton1`, and `jCheckbox1` the same width as `jList1`:

   a Holding down the *Ctrl* key, select all four components, starting with `jList1`.

   b Right-click over one of the selected components and choose Same Size Horizontal.

   Since `jList1` was the first component selected in the group, the other components match its width.

Note While it's actually not necessary to make the components the same width, it is preferable in this case because it simplifies the grid created during the conversion to `GridBagLayout`. This decreases the number of columns generated for `GridBagLayout`, as demonstrated earlier in "Step 1: Design the layout structure" on page 11-17.

Tip For best results, when laying out components in `XYLayout` for conversion to `GridBagLayout`, you should try to match the start and end of components evenly. Wherever possible, try to make the components conform more to an even grid design, rather than a stair-step design.

When the end of a component on one row overlaps the start of another component on a different row, `GridBagLayout` has a more difficult time calculating how many cells to create, which cells should have weights, how many cells the component should span, and how to apply insets and padding. The results are often not quite right, and it can take additional time to clean up.

Create the right panel and add its components

Below is a quick way to create the right panel, since it is pretty much identical to the existing one.

1 Right-click on `jPanel2` in the designer and choose Copy from the context menu.

2 Place the cursor to the right of `jPanel2` in the designer, approximately at the location you want the upper-left corner of `jPanel3` to appear.

3 Right-click and choose Paste. A new panel called `jPanel3` is created, containing `jLabel2`, `jList2`, `jButton2`, and `jCheckbox2`.

4 Change the text property values for `jLabel2`, `jButton2`, and `jCheckbox2` as follows:

`jLabel2` = "Available Columns"
`jButton2` = "Add to Sort"
`jCheckbox2` = "Case Sensitive"

5 Now lets align the two panels vertically. Hold down the *Ctrl* key and select `jPanel2` first, then `jPanel3`. Right-click over one of the selected panels in the designer and choose Align Top from the context menu.

Create the bottom panel and add its components

1 Add the final panel, `jPanel4`. Drag a new `jPanel` component across the entire width of `jPanel1` at the bottom, roughly aligning its left edge with the left edge of `jPanel2` and its right edge with right edge of `jPanel3`.

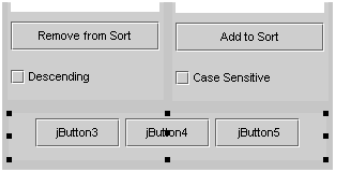
Figure 11.31 Draw bottom panel

2 Right-click `jPanel4` and choose Align Center.

3 Drop three `jButton` components into `jPanel4`.

4 Change the font for these buttons to match the other components.

Figure 11.32 GridBagLayout tutorial, add button panel



Note Since the default layout for a `jPanel` is `FlowLayout`, you can take advantage of this temporarily to add the buttons to the panel. As you drop the buttons into a `FlowLayout` panel, the layout manager centers the buttons in the panel with an even horizontal distance between them. You can then go directly to `GridLayout` without needing to use `XYLayout`

5 Change the `text` property for each of the three buttons to display OK, Cancel, and Help, in that order. The layout manager adjusts the width

of the buttons to fit the text. Don't bother with resizing or positioning them, because when you convert this panel to `GridLayout`, the buttons become the same size, height, and width.

Congratulations! You're all done with the initial layout. Save your file before proceeding.

Step 4: Convert the outer panel to GridBagLayout

You're ready to convert jPanel1 to `GridBagLayout`.

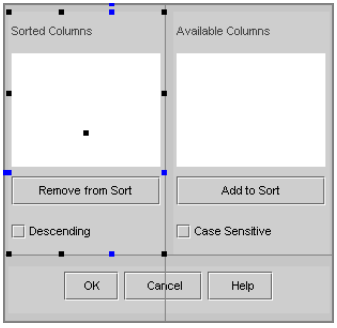
1 First, do a final check to make sure all three panels are aligned well, and that there is a nice amount of space between the inner panels and the edges of `jPanel1`.

Tip For best results, don't crowd the components in their containers, otherwise you may have a mismatch between the collective minimum or preferred size of the components and the minimum or preferred size of the containers, including the `Frame`.

2 Select `jPanel1` and change its `layout` property to `GridBagLayout`. This should result in very little visible change to your layout, especially since you grouped the components nicely into three panels that are easy for the `GridBagLayout` manager to manage.

If you select one of the panels inside the GridBagLayout container, like `jPanel2`, you should see a grid of only two columns and two rows, as shown below:

Figure 11.33 Columns after conversion



Step 5: Convert the upper panels to GridBagLayout

In this design, it really doesn't make a difference whether you convert the main container to `GridBagLayout` first or these two panels. Since the intended layout for `jPanel2` and `jPanel3` is `GridBagLayout`, the size of the panels does not change when they are converted (unlike changing to `GridLayout` or `FlowLayout` which resize to fit their components.)

To change these two panels to GridBagLayout:

1. First, make sure the alignment of the components in these panels is as clean and as simple as possible (the goal being to minimize the number of columns needed for the grid).
2. Select both jPanel2 and jPanel3 and change their layouts to GridBagLayout.

You should see little difference in these panels after you convert them to GridBagLayout, although you might lose some of the margin, or gap, around the outside of the two panels.

Don't worry if things are not perfect yet. We'll make final adjustments in Step 7.

Step 6: Convert the lower panel to GridLayout

The final conversion is to the bottom panel:

1. Select jPanel4 and change its layout to GridLayout.

Note Since jPanel4 was left in FlowLayout during jPanel1's GridBagLayout conversion, and since it was stretched across the entire width of jPanel1, it should have stayed nicely centered in the container after converting to GridBagLayout, spanning all columns. JBuilder assumes for conversion purposes that jPanel1's preferred size is the size it was in XYLayout and assigns component constraints during the conversion based on this pixel size (for example, by giving it a gridwidth value equal to the number of columns generated during the conversion or its ipadx value.)

Now, as you change the layout to GridLayout, the panel stays the same width as it was in XYLayout, but notice that the buttons expand to fill the panel, and there is no gap between them. Also, the panel is larger than necessary. We'll fix these things in Step 7, where we'll make minor adjustments to the constraints for all the components.

2. Save your file now before you start making modifications.

Step 7: Make final adjustments

Rather than just give you the constraint values that make this UI design work, we're going to examine each component separately to show why we made the decisions we did. This approach should give you a much better understanding of how the constraints affect the components, cells, and other components.

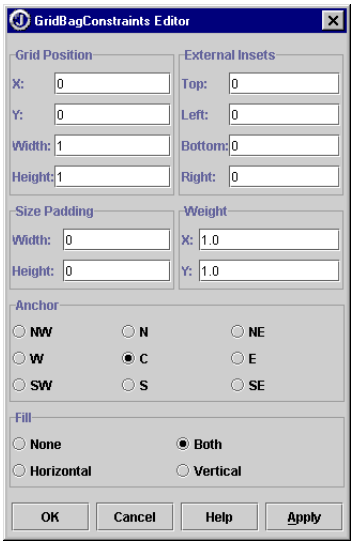
One thing to keep in mind as we continue is that all the discussion about how the constraints affect the components during resizing is relevant only if at least one component has weightx or weighty constraints. In fact, it is

unlikely that you would ever create an outer `GridBagLayout` container without using weight constraints. When none of the components have weight, resizing has no affect on the placement of the components. All the components clump at their minimum or preferred size in the center of the `GridBagLayout` panel and any extra space given to the container by enlarging it gets put between the outside edges of the components and the edge of their container.

Another important point is that there is more than one combination of constraint values that can accomplish the same results. For example, as with the `GridLayout` panel below, to keep it centered and a consistent size at the bottom of the `GridBagLayout` container, you can use `insets`, `anchor`, `padding`, or a combination of these.

Most of the modifications in the rest of this section of the tutorial are made in the GridBagConstraints Editor.

Figure 11.34 GridBagConstraints Editor



To open the GridBagConstraints Editor,

1. Select a component that is inside a `GridBagLayout` container. If the component is a panel containing other components, select it, then move the cursor over the middle nib where a four-headed arrow appears.

Note You can't bring up the GridBagConstraints Editor for `jPanel1` because it's a component inside a `BorderLayout` container. You can only bring up the GridBagConstraints Editor for components inside a `GridBagLayout` container, such as `jPanel2` or `jPanel3`.

2. Right-click and choose Constraints from the context menu.

After finishing this tutorial, do some experimentation with this UI. Open the GridBagConstraints Editor and try out different constraint values to see what happens.

Let's continue by fixing the `GridLayout` panel (`jPanel4`) first.

GridLayout panel

As soon as you converted `jPanel4` to `GridLayout`, the buttons inside it expanded to completely fill the panel with no gaps. Let's first put a little gap between the buttons. It's just a matter of setting a value for the `hgap` property for the `GridLayout` itself.

To change the horizontal gap value,

1. Click `gridLayout1` in the component tree immediately below the `jPanel4` node.
2. Click `gridLayout1`'s `hgap` property in the Inspector and enter a pixel value of 6, then press *Enter*.

Next, the buttons need to be smaller. Since this is a `GridLayout`, the buttons fill up the grid, and if you enlarge the `Frame`, the buttons also expand, as demonstrated below.

Figure 11.35 GridLayout with fill on before resizing

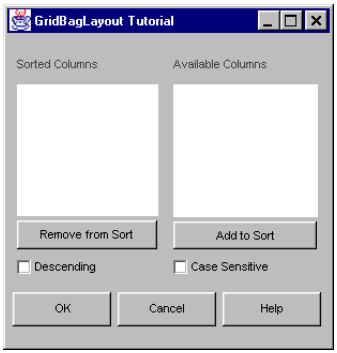
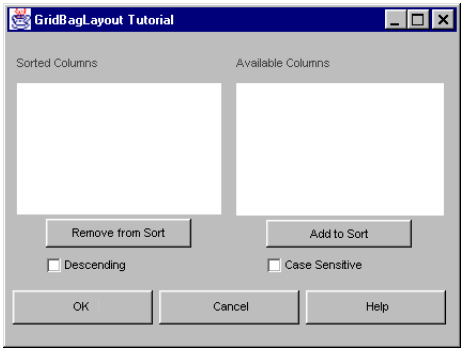


Figure 11.36 GridLayout with fill on after resizing



This is the expected behavior of GridLayout: the components it contains fill up the panel, no matter what size it is (honoring any values specified for horizontal and vertical gap surrounding the buttons.)

Therefore, to control the size of the buttons in the grid, you must restrict the size of the GridLayout panel itself, using its GridBagConstraints.

fill

Select jPanel4 in the component tree, then click the ellipsis button for the constraints property in the Inspector to open the GridBagConstraints Editor. If you look at the constraint values assigned to jPanel4, you'll see that both its horizontal and vertical fill constraints are turned on. When this is the case, GridBagLayout stretches the panel to completely fill its display area, up to the edge of any insets that are set. If there are no insets, the panel fills up the display area to the edge of the cells.

This is definitely not the behavior we want for this GridLayout panel since we want the buttons in the panel to be their preferred size. To accomplish this, you need to remove the panel's fill constraints.

To remove both the horizontal and vertical fill constraints at the same time, check None for the fill constraint value in the GridBagConstraints Editor and click OK.

Figure 11.37 fill constraints

Alternatively, to remove the fill constraints for a component, you can right-click the component where the cursor turns into a double-sided arrow and choose Remove Fill from the context menu.

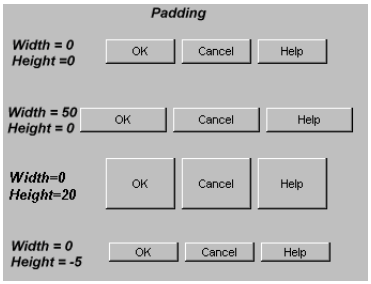
Notice that this action didn't make the buttons shrink to their preferred size. You also need to adjust the padding values to accomplish this.

Padding

Padding (ipadx, ipady) changes the actual size of the panel by adding a specified number of pixels to its minimum width and/or height. The minimum size of the GridLayout panel is just large enough to display the buttons at their minimum size, plus any width you specified in the hgap property for GridLayout. In the case of buttons, there is a margin property that is also included in the calculation of the button's minimum size.

If you like the size of the buttons and the panel at their minimum size, then you don't need to do anything more with the padding. If you want the buttons in the GridLayout panel to be larger than their minimum size, you can specify how many additional pixels to add to the panel to accomplish this. You can even make the buttons smaller by using negative values.

Figure 11.38 Example of different padding values



For this tutorial, since the size of the buttons in `jPanel4` is acceptable at their minimum size, set both padding values to zero.

Figure 11.39 Padding constraints

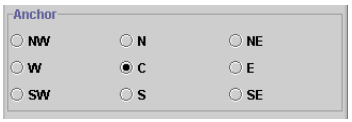
Note Remove padding values for multiple components by multiply selecting the components, right-clicking, and choosing Remove Padding.

Now the buttons shrink to their preferred sizes.

anchor

To make sure the `jPanel4` always stays centered in its display area, you need to set the `anchor` constraint to Center. Since we centered the panel before we converted to `GridBagLayout`, the `anchor` constraint is probably already set to Center. But, open the GridBagConstraints Editor and make sure it looks like the following:

Figure 11.40 anchor constraints



Now that the `fill` is None, the padding values are zero, and the `anchor` is Center, when the container is resized, the buttons stay small and centered when the `Frame` is resized.

insets

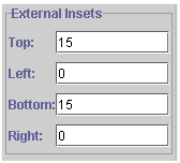
Insets simply define an area between the component and the edges of its display area into which the component cannot enter. It is just like setting margins in a document. No matter how the container is resized, the number of pixels for the `insets` remains constant and work like brakes to keep the component away from the edge of the display area.

If you have `fill` turned on for the component, it fills the display area up to the `insets`. As you resize the container, the component expands to stay up against those `insets`.

In the case of this `GridLayout` panel, since you removed the `fill` and anchored it in the center of its display area, nothing would be gained by adding any `insets` to the left and right edges of the display area. The only thing you need to do here is make sure both the left and right `Inset` values match (set to zero).

You do, however, want to set the top and bottom `insets` to add space above and below `jPanel4`. Set each of these values in the GridBagConstraints Editor to 15 pixels.

Figure 11.41 insets constraints



That takes care of the `GridLayout` panel. Now, lets move on to the upper panels.

Upper panels

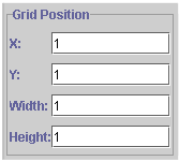
You mostly need to do some clean-up and constraint matching for these panels (`jPanel2` and `jPanel3`) and their components.

gridwidth and gridheight

First, open the GridBagConstraints Editor for each of the components in `jPanel1`, and in the Grid Position area, check that each component only specifies a value of 1 cell for the Width and Height values (`gridwidth` and `gridheight`) . If not, correct this.

Note Do not adjust the X or Y values in the Grid Position area.

Figure 11.42 gridwidth and gridheight



Tip You can modify the constraint values for all the components in a single `GridBagLayout` container at once. Hold down the *Ctrl* key and select all the components, then right-click over one of the selected components and choose Constraints. Change the desired constraints and click Apply or OK.

fill

Next, you want all the components and their containers to fill up their display area, except for the buttons. As in the `GridLayout` panel, we don't want the buttons to expand as the `Frame` is resized.

Again, working with one panel at a time, select each component inside `jPanel2` and `jPanel3`, except the button, and check Both for the `fill` constraint. Right-click the center of each component and use the context menu.

Lastly, you want the panels themselves to fill up their display area in the main `GridBagLayout` container. So, make sure `jPanel2` and `jPanel3` have a `fill` constraint of Both as well.

anchor

The two panels, and their label, list, and checkbox components all have `fill` constraints of Both. Since each of these components fills its display area both horizontally and vertically, `anchor` constraints have no effect. There is simply no room inside the display area for the component to move.

If you want to verify this, try changing some of the `anchor` constraints for these components, then running your program and resizing the container. You'll see there is no change.

The only components in these panels for which `anchor` has an effect are the buttons, which have no `fill`. Since they do not fill up their display area, they can be moved around inside it. To make sure these buttons stay centered in their display area, set their `anchor` constraint to Center.

insets

Since the components in both of these panels happen to be the same, matching `insets` for all of them ensures that the components look the same in both panels. None of the components need left and right `insets`, as the panels containing them are invisible and control the spacing in the main container. However, top and bottom `insets` put some spacing between each of the components within the panels.

Set the `inset` values for the components in `jPanel2` and `jPanel3` in the External Insets area of the GridBagConstraints Editor as follows:

Labels:	Top = 0, Left = 0, Bottom = 4, Right = 0
Lists:	Top = 0, Left = 0, Bottom = 0, Right = 0
Buttons:	Top = 10, Left = 10, Bottom = 0, Right = 10
Checkboxes:	Top = 6, Left = 0, Bottom = 0, Right = 0

Finally, to put a little space between the top and sides of these two panels and the outer container (`jPanel1`), set the following `insets` for `jPanel2` and `jPanel3`:

Top = 10, Left = 10, Bottom = 0, Right =10

Note You don't need to set `insets` for the bottom, since the `GridLayout` panel's top `insets` are taking care of that space.

ipadx, ipady

One place in this design where `ipadx` (horizontal padding) is appropriate is for controlling the size of the Add to Sort button in `jPanel3`. If you leave the `ipadx` value at zero for both buttons, then the Add to Sort button displays at its minimum size which won't match the size of the Remove from Sort button.

You can use horizontal padding (`ipadx`) to increase the width of the button and make it the same width as the other button.

1. Select the Remove from Sort button and open the GridBagConstraints Editor. Make sure the Padding Width and Height values are set to zero, and the `fill` constraints still say None.
2. Select the Add to Sort button, and type in a pixel value of 33 for the Padding Width. This amount made the buttons match width in our example UI. If it has a different result for you, experiment with different values until you find one that works.

Figure 11.43 Add to Sort button without ipadx

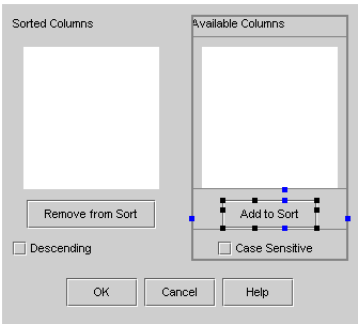
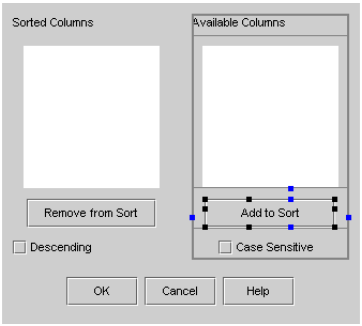


Figure 11.44 Add to Sort button with ipadx



Tip If changing `ipadx` to zero didn't make the Remove from Sort button wide enough to display all the text, you may also want to select 'this' in the component tree to expose the nibs for the main container, and drag the right nib to widen the container a bit.

Another thing you could do is make these buttons a bit smaller vertically than their preferred size by using a negative value in Padding Height (we used a -3). This is, of course, personal preference.

None of the other components in these panels need padding, nor do the panels themselves. Since the `jPanel2` and `jPanel3` have `fill` constraints, these override any padding that might be assigned.

You'll also notice that the list components use `ipadx` and `ipady`, which you might not want in reality. Since you did nothing to populate the lists with items in this example, if you remove the `ipadx` and `ipady` constraints along with the `fill`, the lists disappear. Their minimum size is determined by the number of items in the list. We just added `ipadx` and `ipady` constraints to force a particular size for demonstration purposes.

weightx and weighty

As we said earlier, if you want the components in a container to change size as the container is resized, you need to assign weightx and/or weighty constraint values to at least one component horizontally and vertically. Weight constraints specify how to distribute the extra container space created when resizing the container.

You need to set both the `weightx` and/or `weighty` constraints, plus the `fill` constraints for a component if you want it to grow. For example, if a component has a horizontal weight constraint (`weightx`), 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 `weightx` (or `weighty`) 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).

First, we took all the weight constraint values off that the conversion had set. This is what happened:

Figure 11.45 No weight constraints, before resizing

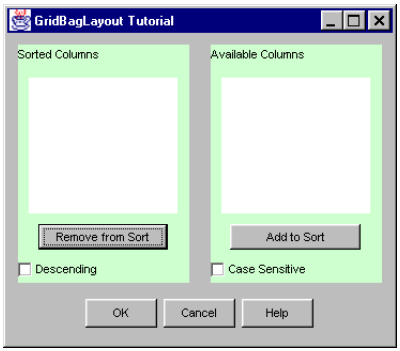
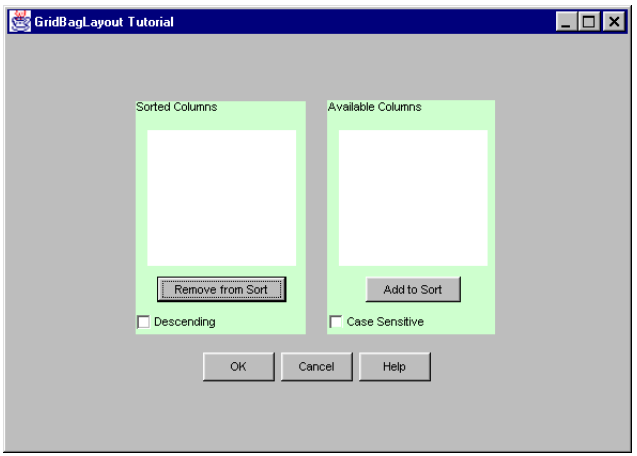


Figure 11.46 No weight constraints, after resizing



Notice how the components are all clumped in the middle after resizing.

We determined that the components we want to grow are `jPanel2` and `jPanel3`, and both list components inside them. We tried various weight constraint combinations on these components to see the results. The following list points to these results:

- Weight constraints on both panels and lists
- Weight constraints on the panels, but not on the lists
- Weight constraints on the lists, but not on the panels
- Horizontal weight constraints only
- Vertical weight constraints only
- Weight constraints on only one panel and list component in the row

We want weight constraints on both panels and lists. Set both the weight constraints to 1.0 in the GridBagConstraints Editor for all four components: `jPanel2`, `jPanel3`, `jList1`, and `jList2`.

Conclusion

Congratulations! You've completed this tutorial, and have a better understanding of `GridBagLayout` and the function of each of the `GridBagConstraints`.

One thing that should be obvious from this exercise is that each `GridBagLayout` is going to require experimentation with the constraints until you achieve just the look and behavior you want. JBuilder can assist in that process by quickly getting you past the initial `GridBagLayout` coding and on to the fine tuning.

`GridBagLayout` is a powerful tool, but not necessarily an easy one to use. Keep in mind that, just like anything else, the more you practice, the easier it gets.

Part 3: Tips and techniques

XYLayout is a feature of JBuilder SE and Enterprise. If you use JBuilder Personal, substitute null layout wherever XYLayout is specified.

Setting individual constraints in the designer

anchor

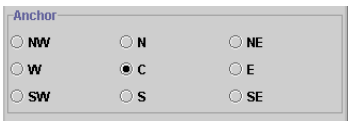
There are two ways to set a component's `anchor` constraint in the designer:

- Click the component and drag it toward the desired location at the edge of its display area, much like you would dock a movable toolbar.

  For example, to anchor an image in the upper left corner of its display area, click the mouse in the middle of the image and drag it until the upper left corner of the image touches the upper left corner of the display area. This sets the `anchor` constraint value to `NW`, both in the GridBagConstraints Editor and in the code.

- Select an `anchor` constraint value in the GridBagConstraints Editor.

Figure 11.47 Anchor constraints in GridBagConstraints Editor



To do this,

a Select the component on the design surface.

b Right-click the component and choose Constraints to open the GridBagConstraints Editor.

c Click the desired value in the `anchor` area, then press OK.

This changes the constraint value in the code and relocates the component to its new anchor point on the design surface.

Note Moving the component with the mouse updates the `anchor` constraint value in the GridBagConstraints Editor. Similarly, when you change the constraint value in the GridBagConstraints Editor, the component moves to its new location on the design surface.

fill

The fastest way to specify the `fill` constraint for a component is to use the component's context menu on the design surface.

1. Right-click the component on the design surface to display the context menu.
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 (this requires displaying the context menu twice).
   - Select Remove Fill to set the value to NONE.

You can also specify the `fill` constraint in the GridBagConstraints Editor.

1. Right-click the component on the design surface and choose Constraints to display the GridBagConstraints Editor.
2. Select the desired constraint value in the Fill area, then press OK.

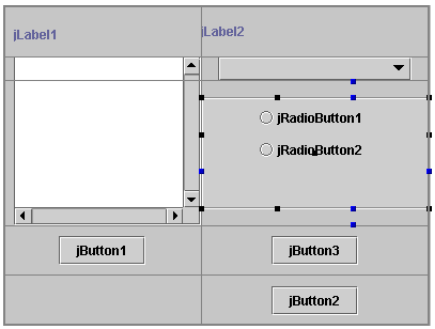
Figure 11.48 Fill constraints in the GridBagConstraints Editor

insets

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.

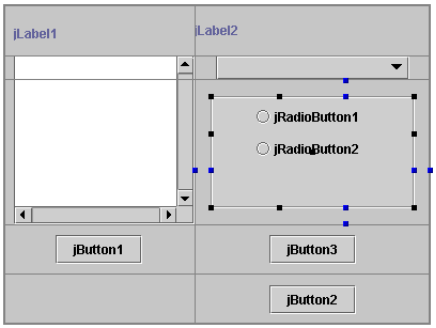
When an Inset value is zero, you only see one blue nib on that side of the cell, as shown below.

Figure 11.49 Insets set to zero on top and bottom of component



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 (in pixels) between the two nibs. Grab either nib to change the size of the Inset.

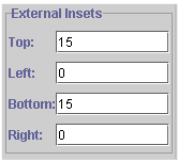
Figure 11.50 Insets greater than zero on right and left sides of component



For more precise control over the Inset values, use the GridBagConstraints Editor to specify the exact number of pixels.

1. Right-click the component on the design surface and choose Constraints to display the GridBagConstraints Editor.
2. In the External Insets area, specify the number of pixels for each Inset: Top, Left, Bottom, or Right.

Figure 11.51 Insets in the GridBagConstraints Editor



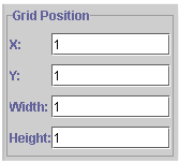
Note While negative Inset values are legal, they can cause components to overlap adjacent components and are not recommended.

gridwidth, gridheight

You can specify `gridwidth` and `gridheight` constraint values in the GridBagConstraints Editor.

1. Right-click the component on the design surface and choose Constraints to display the GridBagConstraints Editor.
2. 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 occupies in the row or column.

Figure 11.52 Gridwidth and gridheight in the GridBagConstraints Editor



- If you want the value to be RELATIVE, enter a -1.
- If you want the value to be REMAINDER, enter a 0.

Note JBuilder never generates a `gridwidth` or `gridheight` value of REMAINDER during conversion to `GridBagLayout`.

You can also use the mouse to change the `gridwidth` or `gridheight` by sizing the component into adjacent empty cells (dragging a black sizing nib across the cell border.)

ipadx, ipady

You can specify a component's padding (`ipadx` or `ipady`) values by clicking on any of the black sizing nibs at the edges of the component, and dragging with the mouse to increase or decrease the size of the component. If you make the component larger than its preferred size, you see a positive pixel value. If you make the component smaller than its preferred size, you see a negative pixel value.

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).

Figure 11.53 Before: jButton2 gridwidth = 1 cell, ipadx = 46

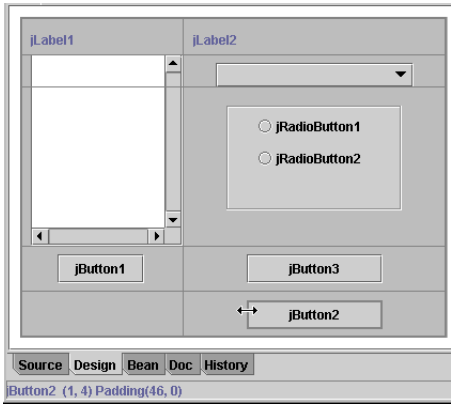
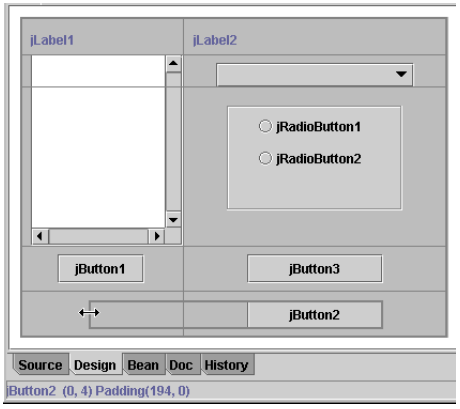


Figure 11.54 After: jButton2 gridwidth = 2 cells, ipadx increased to 194



For more precise control over the `ipadx` and `ipady` values, use the GridBagConstraints Editor to specify the exact number of pixels to use for the value.

1. Right-click the component on the design surface and choose Constraints to display the GridBagConstraints Editor.
2. In the Size Padding area, specify the number of pixels for the Width and Height values.

Figure 11.55 Size Padding in the GridBagConstraints Editor

Note Negative values make the component smaller than its preferred size and are perfectly valid.

To quickly remove the `ipadx` and `ipady` constraints (set them to zero), right-click the component on the design surface 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.

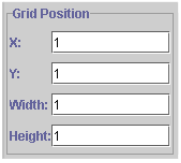
gridx, gridy

You can use the mouse to specify which cell the upper left corner of the component occupies. 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. Right-click the component on the design surface and choose Constraints to display the GridBagConstraints Editor.
2. In the Grid Position area, enter the number of columns for the X value, or the number or rows for the Y value. If you want the value to be RELATIVE, enter a -1.

Figure 11.56 Grid Position in the GridBagConstraints Editor



Tip As you move the component on the design surface, the `gridx` (column) and `gridy` (row) positions are displayed and updated in the status bar at the bottom right. "col:" is `gridx` and "row:" is `gridy`. The values in the GridBagConstraints Editor are updated as well.

Note When you use the mouse to move a component to an occupied cell, the 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.

weightx, weighty

If you want your cells to grow, `weightx` and `weighty` must be set to a non-zero value.

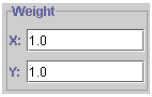
To specify the weight constraints for a component on the design surface, right-click the component and choose Weight Horizontal or Weight Vertical. This sets the value to 1.0.

To remove the weight constraints (set them to zero), right-click the component and choose Remove Weights. You can do this for multiple components in a container: hold down the *Shift* key when selecting the components, then right-click and choose Remove Weights.

If you want to set the weight constraints to something other than 0.0 or 1.0, you can set the values in the GridBagConstraints Editor.

1. Right-click the component(s) and choose Constraints to display the GridBagConstraints Editor.
2. Enter a value between 0.0 and 1.0 for the X or Y value in the Weight area, then press OK.

Figure 11.57 Weight constraints in the GridBagConstraints Editor



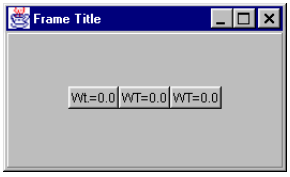
Important `weight` constraints can sometimes make the sizing behavior in the designer difficult to predict. Therefore, setting these constraints should be the last step in designing `GridBagLayout`.

Behavior of weight constraints

Below are some examples of how weight constraints affect the behavior of components:

- 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 don't expand beyond their preferred size. `GridBagLayout` puts any extra space between its grid of cells and the edges of the container.

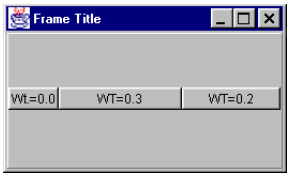
  Figure 11.58 All components have weight constraints of zero in the same direction

- If you have three components with `weightx` constraints of 0.0, 0.3, and 0.2 respectively, when the container is enlarged, none of the extra space

goes to the first component, 3/5 of it goes the second component, and 2/5 of it goes to the third.

Figure 11.59 Three components with weightx constraints of 0.0, 0.3, and 0.2 respectively



- You need to set both the weight and `fill` constraints for a direction (vertical or horizontal) of a component if you want it to grow.

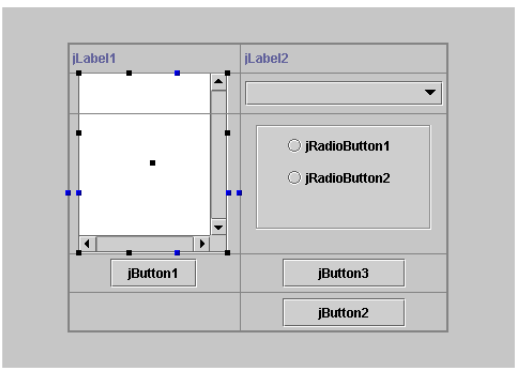
  For example,

  - If a component has a non-zero `weightx` constraint value, 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 display area. It enlarges the width of the display area without changing the size of the component.
  - If a component has both weight and `fill` constraints, then the extra space is added to the display area, plus the component expands to fill the new display area 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 weight constraint values 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 `ipadx` and `ipady` constraints.

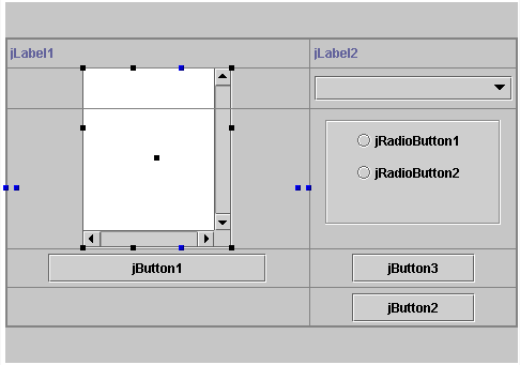
    Figure 11.60 GridBagLayout with zero weight constraints on all components



When this frame is resized, the components remain at their preferred size clumped in the middle, and the space around them grows to fill the enlarged container.

In the next example, `jScrollPane1` has a horizontal weight (`weightx`) constraint of 1.0. Notice that as soon as one component in the container is assigned any `weight`, the components are no longer centered in the panel. Since a `weightx` constraint is set, the `GridBagLayout` manager takes the extra space in the `GridBagLayout` panel that was previously on each side of the grid and puts it into the cell containing `jScrollPane1`. Also notice that `jScrollPane1` does not change size from the previous example.

Figure 11.61 jScrollPane1 with weightx=1.0, no fill constraints

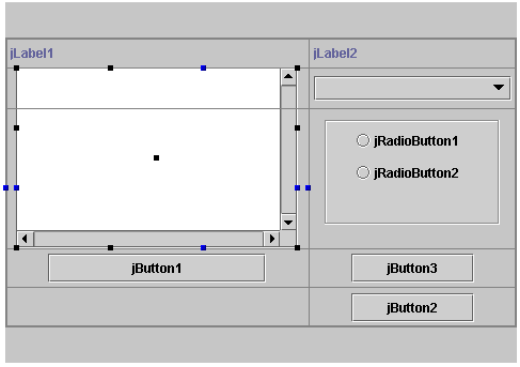


Note

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` frame in the component tree, then click on its black sizing nibs and drag the frame to the desired size.

In the final example, `jScrollPane1` has both a `weightx` constraint of 1.0 and a horizontal `fill` constraint. Notice that `jScrollPane1` expands to fill the width of the display area.

Figure 11.62 jScrollPane1 with weightx=1.0, fill=Horizontal



Important If one component in a column has a `weightx` value, `GridBagLayout` gives the whole column that value. Conversely, if one component in a row has a `weighty` value, the whole row is assigned that value.

Using drag and drop to edit constraints

When you drag a component in the designer, the outline of the component indicates where the component will drop. The status bar indicates which column and row the component is in and if the target cell is occupied, what new column and/or row will be created to accommodate the component in its new position.

Figure 11.63 Component being dragged in current cell in the designer

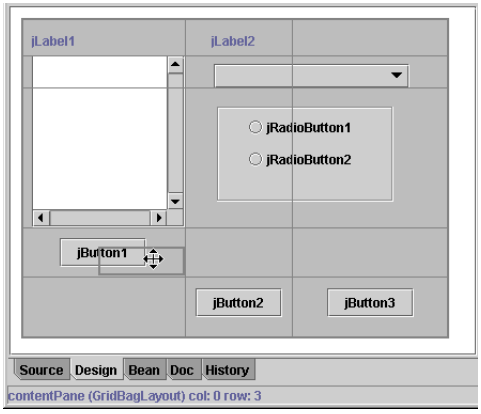
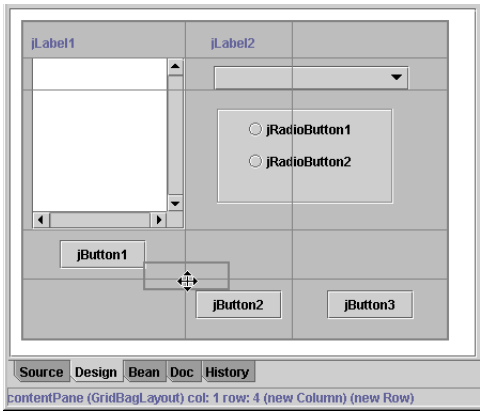


Figure 11.64 Component being dragged to new, occupied cell in the designer



Also, when you drag a component to a new location in its display area, JBuilder determines the nearest `anchor`, then applies the `insets` necessary to make the component stay where you put it.

Dragging a component to an empty cell

When dragging a component into an empty cell, JBuilder retains the component's `insets` and `fill` constraints settings.

For example, if the `fill` constraint is set to Horizontal, when you move the component to a wider cell, `GridBagLayout` expands the component to fill the new cell.

Figure 11.65 Before moving jButton2, with fill constraint set to HORIZONTAL

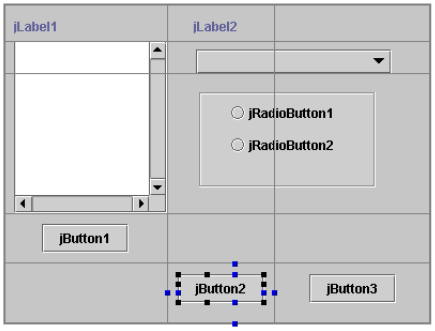
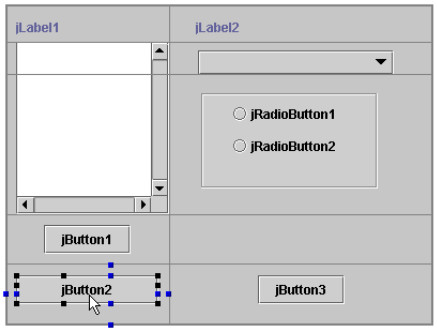


Figure 11.66 Moving jButton2 to same size or larger cell



Notice that in this case, moving `jButton2` to the empty cell under `jButton1` also causes one of the columns to be removed from the grid since it is no longer needed for any components. The constraints values for other components are automatically modified as needed to accommodate this change (for instance `gridwidth` changes from 2 to 1 for components that previously spanned both column 1 and column 2).

Note The first column in the grid is number 0.

This a good example, however, of why modifying a design once the container is converted to `GridBagLayout` can be tricky, since you can experience unexpected results.

If the component you're moving is larger in a dimension than the cell into which it is being moved, `GridBagLayout` spans the component across two cells, then applies the `fill` constraints and `insets`.

For example, if the `fill` constraint for `jButton3` is set to HORIZONTAL and you drag it to the cell above `jButton2` which is smaller than the button's current size, `GridBagLayout` makes the button span both column 1 and 2.

Figure 11.67 Before moving jButton3, with fill constraint set to HORIZONTAL

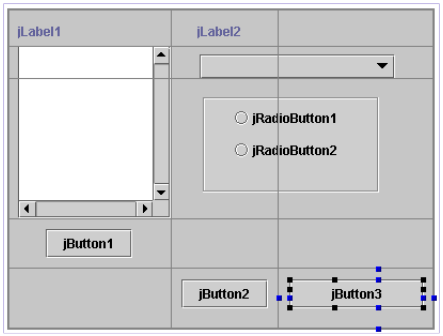
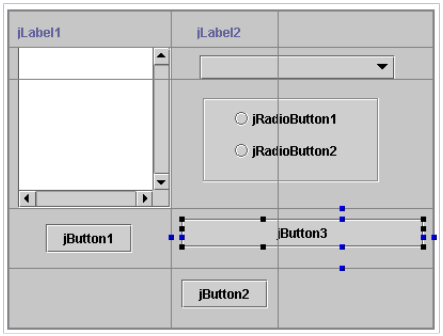


Figure 11.68 Moving jButton3 to smaller cell above jButton2



See "Dragging a large component into a small cell" on page 11-53.

Dragging a component to an occupied cell

This action gives you the most drastic results when doing drag-and-drop editing because only one component can occupy a cell. Therefore, if you try to move a component to an occupied cell, `GridBagLayout` makes other arrangements for the move. It creates a new column or a new row for the component.

The side effect of this is that to accommodate the new cells, it changes the grid position constraints of many of the other components as well. Their `gridx` or `gridy` constraints change if their position is after or below the new columns or rows in the grid. Their `gridwidth` or `gridheight` may change if the already spanned columns or rows are affected by the new ones.

It's especially important to understand this action and resulting behavior. If you are using the designer to build up a new UI in `GridBagLayout`, rather than in `XYLayout` or `null` layout, much of the time the grid will be full when you drop a new component onto it, resulting in the creation of new columns or rows and changed constraints for existing components.

The examples below demonstrate what happens when `jButton3` is dragged to the adjacent cell occupied by `jButton1`. Notice that as `jButton3` is dragged into the cell with `jButton1`, the status bar indicates what column or row the mouse cursor is in and if a new column or row will be created when the button is dropped there.

`GridBagLayout` makes the following changes when `jButton3` is dragged into the same cell as `jButton1`:

- It inserts a new column between the existing two columns.
- It increases the `gridwidth` for `jLabel1`, the `jScrollPane1`, and `jButton2` to two columns instead of one.
- It removes the right `inset` from `jButton1` and the left `inset` from `jButton3`.

Figure 11.69 Dragging jButton3 into cell with jButton1

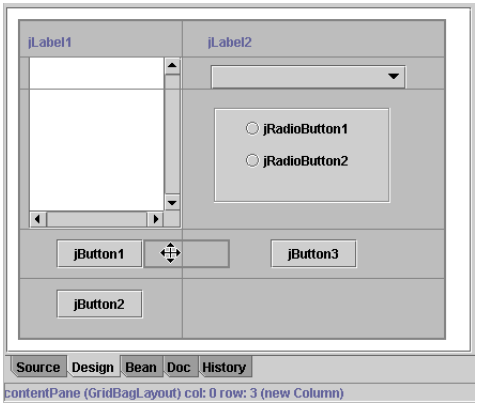
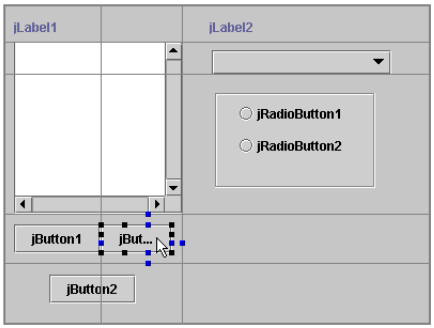


Figure 11.70 After moving the button horizontally



If `jButton3` is dragged into the cell with the radio button panel (`jPanel1`):

- It inserts a new row before `jButton1` by splitting row 2 in half (occupied by the `jPanel1` and the bottom part of the jScrollPane1).
- It increases the `gridheight` of the `jScrollPane1` from two to three rows.
- It takes away space from the large row occupied by the `jScrollPane1` and the `jPanel1`.
- It removes the bottom `inset` for `jPanel1` and the top inset for `jButton3`.

Figure 11.71 Dragging jButton3 into cell with jPanel1

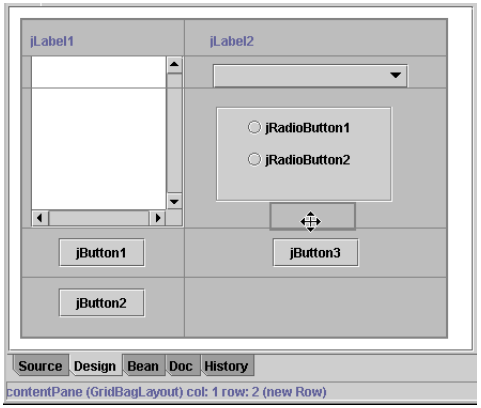
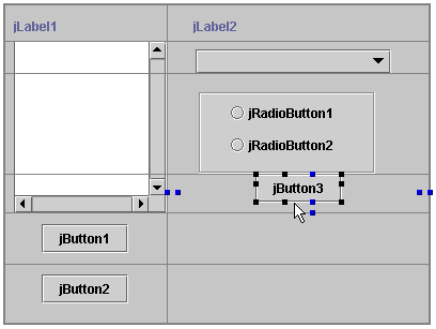


Figure 11.72 After moving the button vertically



Note You get the same results if you drop a new button into the cell occupied by `jButton1` or `jPanel1`, except new components have no `fill` or `insets`, while a moved component retains its `fill` and `insets` constraint settings.

Dragging a large component into a small cell

If you drag a large component into a cell that is smaller than the component, the component spans as many empty cells as it needs, while retaining its `fill` and `insets` constraints. If the component needs more cells than are available (for example, if it runs up against an occupied cell or the edge of the container), then the last cells occupied grow to accommodate the rest of the component, including its `insets`.

Figure 11.73 Dragging jPanel1 into a small, empty cell

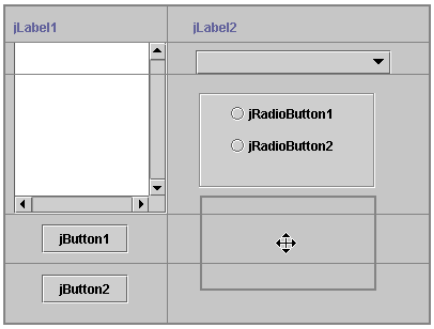
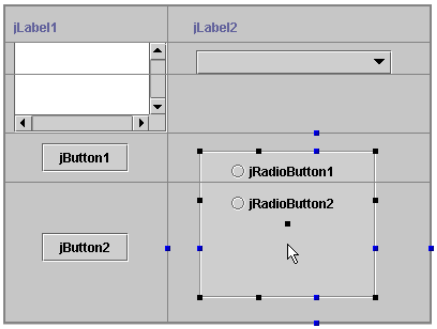


Figure 11.74 After dragging jPanel1



Dragging the black sizing nibs into an adjacent empty cell

Dragging a component's black sizing nib into an adjacent empty cell increases its display area (cell width or height) by one cell in the direction of the move.

Figure 11.75 Before dragging top sizing nib of jButton2 into empty cell above

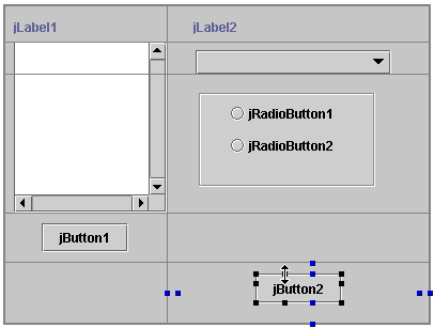


Figure 11.76 During drag

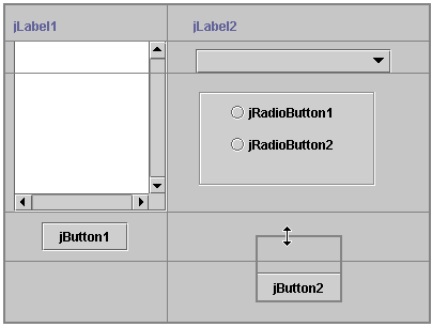
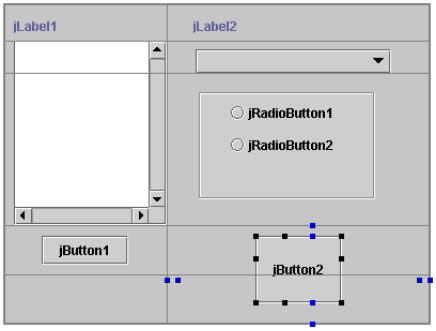


Figure 11.77 After drag, jButton2 has gridheight of 2 with insets unchanged



Dragging the black sizing nibs into an adjacent occupied cell

Dragging a component's black sizing nib into an adjacent occupied cell increases the component's `ipadx` and `ipady` constraint values. Notice how the status bar indicates this in the example below.

Figure 11.78 Before dragging, jButton2 has zero padding

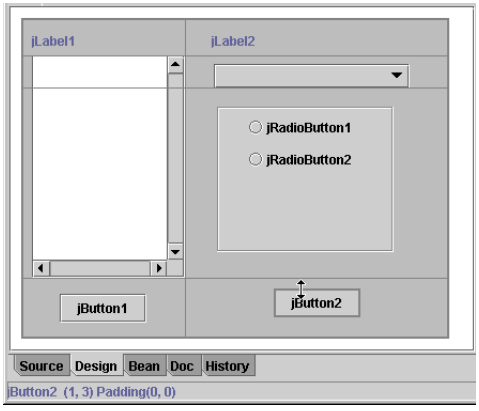


Figure 11.79 Dragging sizing nib into occupied cell increases padding

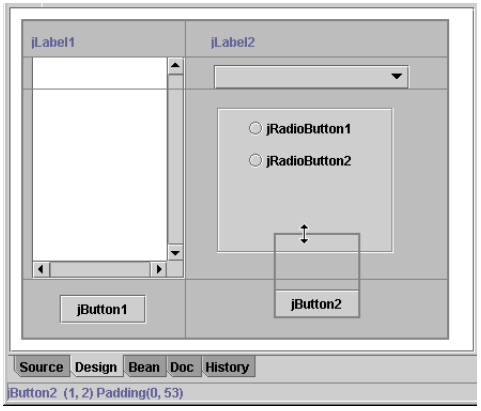
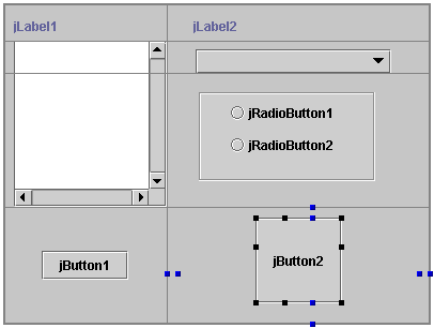


Figure 11.80 After dragging, cells are larger since padding is increased



Adding components

When you add a new component to a GridBagLayout container, where you click to drop the component determines what new columns or rows are created to accommodate it.

Note The default `fill` and `insets` constraints for a new component being added to GridBagLayout are `None`.

- To create a new row above an existing component, click at the top of the cell containing that component.
- To create a new row below an existing component, click at the bottom of the cell containing that component.
- To create a new column to the left of an existing component, click at the left of the cell containing that component.
- To create a new column to the right of an existing component, click at the right of the cell containing that component.

Once you have selected a component on the component palette, watch the status bar as you move your mouse over the grid to see what to expect. The status bar indicates what column and row the component will occupy (its `gridx` and `gridy` position), as well as what new column or row will be created to accommodate the component.

The following examples demonstrate adding a new component on each side of `jButton1`:

Figure 11.81 Clicking above jButton1 creates a new row for jButton3 above jButton1

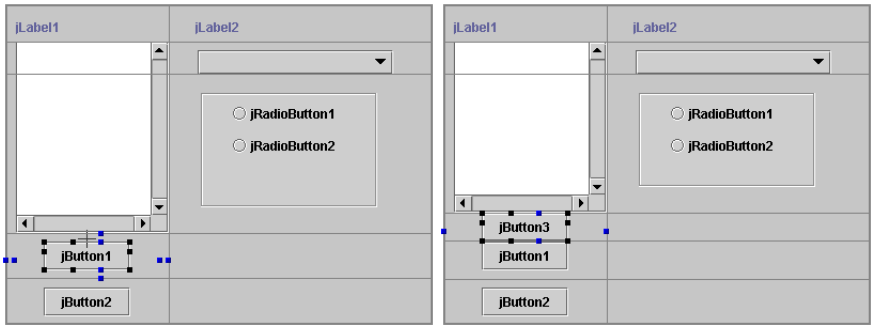


Figure 11.82 Clicking to the left of jButton1 creates a new column for jButton3 to the left of jButton1

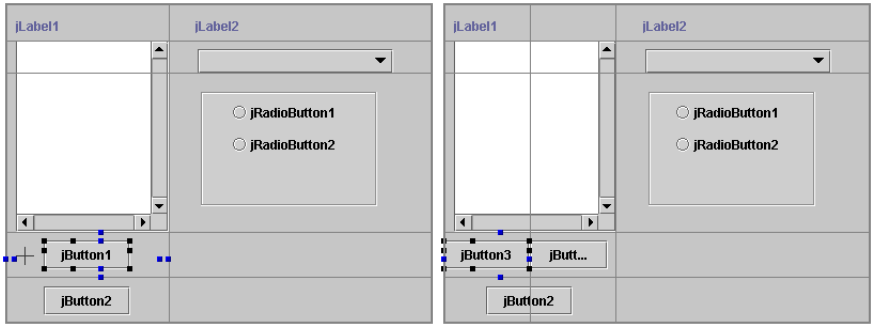


Figure 11.83 Clicking below jButton1 creates a new row for jButton3 below jButton1

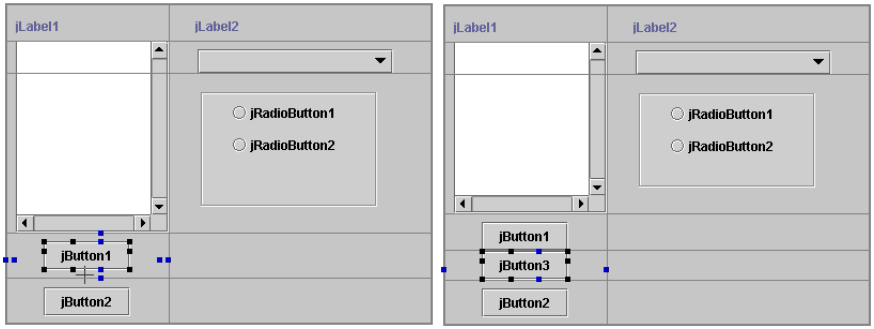
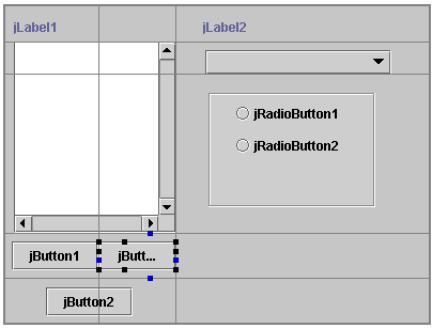


Figure 11.84 Clicking to the right of jButton1 creates a new column for jButton3 to the right of jButton1



Miscellaneous tips

Switch back to XYLayout for major adjustments

If your conversion doesn't give you what you expect, or if you need to add additional components to the design, switch back to `XYLayout`, make the changes, then re-convert to `GridBagLayout`. This may be faster and easier than trying to add components to an existing `GridBagLayout`. Let JBuilder do the work of calculating and assigning the constraints.

Remove weights and fill before making adjustments

It is likely that after conversion to `GridBagLayout`, some adjustments might be necessary. During the conversion process from `XYLayout` to `GridBagLayout`, JBuilder automatically assigns weight constraint values to some of the components.

If you have difficulties as you start moving components or sizing nibs on the design surface, do an Undo, then remove the weight constraint values from all the components in the `GridBagLayout` container. weight constraints are the main culprit in causing unexpected behavior when moving and resizing components graphically in a `GridBagLayout` design. If you remove all weight constraints first, it is easier to make the correct adjustments to the other constraints.

Note This can also be true of `fill` constraints. Removing them may make adjustments easier.

Adjust any other constraints that need modification. When all other constraints are the way you want, then add weight constraints last to only the components that need them.

Making existing GridBagLayout code visually designable

Differences in code

If you create a `GridBagLayout` container by coding it manually, you typically create only one `GridBagConstraints` object for the `GridBagLayout` container and reuse it as you add components to the container. If you want the component you're adding to the container to have different values for particular constraints than the previously added component, then you only need to change those constraint values for use with the new component. These new values stay in effect for subsequent components unless, or until, you change them again.

Important While this method of coding `GridBagLayout` is the leanest (recycling the `GridBagConstraints` object from previously added components), it doesn't allow you to edit that container visually in JBuilder's designer.

When you design a `GridBagLayout` container in the designer, JBuilder creates a new `GridBagConstraints` object for each component you add to the container. The `GridBagConstraints` object has a constructor that takes all eleven properties of `GridBagConstraints` so the code generated by the designer can always follow the same pattern.

```
public GridBagConstraints(int gridx,
                          int gridy,
                          int gridwidth,
                          int gridheight,
                          double weightx,
                          double weighty,
                          int anchor,
                          int fill,
                          Insets insets,
                          int ipadx,
                           int ipady)
```

For example,

```
jPanel1.add(jButton1, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
   GridBagConstraints.CENTER, GridBagConstraints.NONE,
   new Insets(0, 0, 0, 0), 0, 0));
```

Modifying code to work in the designer

If you have a `GridBagLayout` container that was previously coded manually using one `GridBagConstraints` object for the container, before you can design the container visually in JBuilder, you must make the following modification to your code:

For each component added to the container, you must create a `GridBagConstraints` object with a large constructor that has parameters for each of the eleven constraint values, as shown above

Code generated by JBuilder in Part 2

Below is the actual code JBuilder generated when we created the `GridBagLayout` UI in Part2.

```
package gbl;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//import com.borland.jbcl.layout.*;

public class Frame1 extends JFrame {

  //Construct the frame
  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();
  JLabel jLabel2 = new JLabel();
  JButton jButton2 = new JButton();
  JPanel jPanel3 = new JPanel();
  JCheckBox jCheckBox2 = new JCheckBox();
  JList jList2 = new JList();
  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();

  public Frame1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try  {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
//Component initialization

  private void jbInit() throws Exception  {
    this.getContentPane().setLayout(borderLayout1);
    this.setSize(new Dimension(332, 304));
    jPanel2.setBackground(new Color(192, 192, 255));
    jLabel1.setText("Sorted Columns");
    jLabel1.setFont(new Font("Dialog", 0, 11));
    jButton1.setText("Remove from Sort");
    jCheckBox1.setText("Descending");
```

```
    jLabel2.setFont(new Font("Dialog", 0, 11));
    jButton2.setText("Add to Sort");
    jPanel3.setBackground(new Color(192, 192, 255));
    jPanel3.setLayout(gridBagLayout3);
    jCheckBox2.setText("Case Sensitive");
    jButton3.setText("Cancel");
    jButton4.setText("Help");
    jButton5.setText("OK");
    gridLayout1.setHgap(6);
    jPanel4.setLayout(gridLayout1);
    jLabel2.setText("Available Columns");
    jPanel2.setLayout(gridBagLayout2);
    jPanel1.setLayout(gridBagLayout1);
    this.setTitle("Frame Title");
    this.getContentPane().add(jPanel1, BorderLayout.CENTER);
    jPanel1.add(jPanel2, new GridBagConstraints(0, 0, 1, 1, 1.0, 1.0,
       GridBagConstraints.CENTER, GridBagConstraints.BOTH,
       new Insets(10, 10, 0, 10), 0, 0));
    jPanel2.add(jLabel1, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
       GridBagConstraints.WEST, GridBagConstraints.BOTH,
       new Insets(0, 0, 2, 0), 0, 4));
    jPanel2.add(jList1, new GridBagConstraints(0, 1, 1, 1, 1.0, 1.0,
       GridBagConstraints.WEST, GridBagConstraints.BOTH,
       new Insets(0, 0, 0, 0), 128, 128));
    jPanel2.add(jButton1, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0,
       GridBagConstraints.CENTER, GridBagConstraints.NONE,
       new Insets(7, 0, 0, 0), 0, 0));
    jPanel2.add(jCheckBox1, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
       GridBagConstraints.WEST, GridBagConstraints.BOTH,
       new Insets(6, 0, 0, 0), 0, 0));
    jPanel1.add(jPanel3, new GridBagConstraints(1, 0, 1, 1, 1.0, 1.0,
       GridBagConstraints.CENTER, GridBagConstraints.BOTH,
       new Insets(10, 10, 0, 10), 0, 0));
    jPanel3.add(jLabel2, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
       GridBagConstraints.WEST, GridBagConstraints.BOTH,
       new Insets(0, 0, 2, 0), 0, 4));
    jPanel3.add(jList2, new GridBagConstraints(0, 1, 1, 1, 1.0, 1.0,
       GridBagConstraints.CENTER, GridBagConstraints.BOTH,
       new Insets(0, 0, 0, 0), 128, 128));
    jPanel3.add(jButton2, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0,
       GridBagConstraints.CENTER, GridBagConstraints.NONE,
       new Insets(7, 0, 0, 0), 32, 0));
    jPanel3.add(jCheckBox2, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
       GridBagConstraints.WEST, GridBagConstraints.BOTH,
       new Insets(6, 0, 0, 0), 0, 0));
    jPanel1.add(jPanel4, new GridBagConstraints(0, 1, 2, 1, 0.0, 0.0,
       GridBagConstraints.CENTER, GridBagConstraints.NONE,
       new Insets(12, 59, 12, 59), 0, 0));
    jPanel4.add(jButton5, null);
    jPanel4.add(jButton3, null);
    jPanel4.add(jButton4, 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);
    }
  }
}
```

Other resources on GridBagLayout

- java.awt.GridBagConstraints.html at
  http://java.sun.com/j2se/1.3/docs/api/java/awt/GridBagConstraints.html
- java.awt.GridBagLayout.html at
  http://java.sun.com/j2se/1.3/docs/api/java/awt/GridBagLayout.html

GridBagConstraints

anchor

Description:

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`

Valid values:

```
GridBagConstraints.CENTER
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.EAST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST
GridBagConstraints.NORTHWEST
```

Default value:

```
GridBagConstraints.CENTER
```

fill

Description:

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` constraints only affect the component within its own display area. `fill` constraints tell the layout manager to expand the component to fill the whole area it has been given.

Valid values:

`GridBagConstraints.NONE`	Don't change the size of the component.
`GridBagConstraints.BOTH`	Resize the component both horizontally and vertically to fill the area completely.
`GridBagConstraints.HORIZONTAL`	Only resize the component to fill the area horizontally.
`GridBagConstraints.VERTICAL`	Only resize the component to fill the area vertically.

Default value:

`GridBagConstraints.NONE`

insets

Description:

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`.

Valid values:

```
insets = new Insets(n,n,n,n)
```

Top, left, bottom, right (where each parameter represents the number of pixels between the display area and one edge of the cell.)

Default values:

```
insets = new Insets(0,0,0,0)
```

gridwidth, gridheight

Description:

Use `gridwidth` and `gridheight` 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.

Valid values:

`gridwidth=nn`, `gridheight=nn`	Where `nn` is an integer representing the number of cell columns or rows.
`GridBagConstraints.RELATIVE (-1)`	Specifies that this component is the next to last one in the row (`gridwidth`) or column (`gridheight`.) A component with a GridBagConstraints.RELATIVE takes all the remaining cells except the last one. For example, in a row of six columns, if the component starts in column number 3, a gridwidth of RELATIVE makes it take up columns 3, 4, and 5. Note that columns and rows begin numbering at 0 in the grid.
`GridBagConstraints.REMAINDER (0)`	Specifies that this component is the last one in the row (`gridwidth`) or column (`gridheight`).

Default value:

`gridwidth=1`, `gridheight=1`

ipadx, ipady

Description:

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 is 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 is at least the minimum height plus `ipady` pixels.

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.

Example:

When added to a component that has a minimum size of 30 pixels wide 20 pixels high:

- If `ipadx`= 4, the component is 34 pixels wide.
- If `ipady`= 2, the component is 22 pixels high.

Valid values:

`ipadx=nn`, `ipadx=nn`

Default value:

`ipadx=0`, `ipady=0`

gridx, gridy

Description:

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 (top left) cell of the grid.

`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.

Valid values:

`gridx=nn, gridy=nn`

`GridBagConstraints.RELATIVE (-1)`

Default value:

`gridx=1, gridy=1`

weightx, weighty

Description:

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 specified numerically in the range 0.0 to 1.0 inclusive. Zero means the component should not receive any of the extra space, and 1.0 means the component gets a full share of the space.

- The weight of a row is calculated to be the maximum `weightx` of all the components in the row.
- The weight of a column is calculated to be the maximum `weighty` of all the components in the column.

Important If you want your cells to grow, `weightx` and `weighty` must be set to a non-zero value.

Valid values:

`weightx=n.n`, `weighty=n.n`

Default value:

`weightx=0.0`, `weighty=0.0`

Examples of weight constraints

Figure 11.85 Weight constraints on both panels and lists

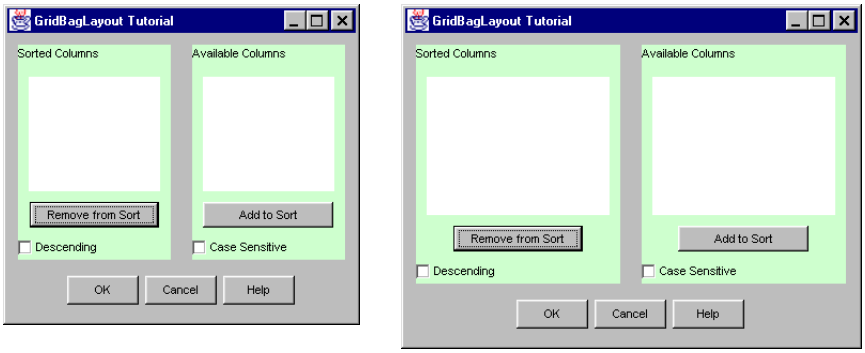


Figure 11.86 Weight constraints on the panels, but not on the lists

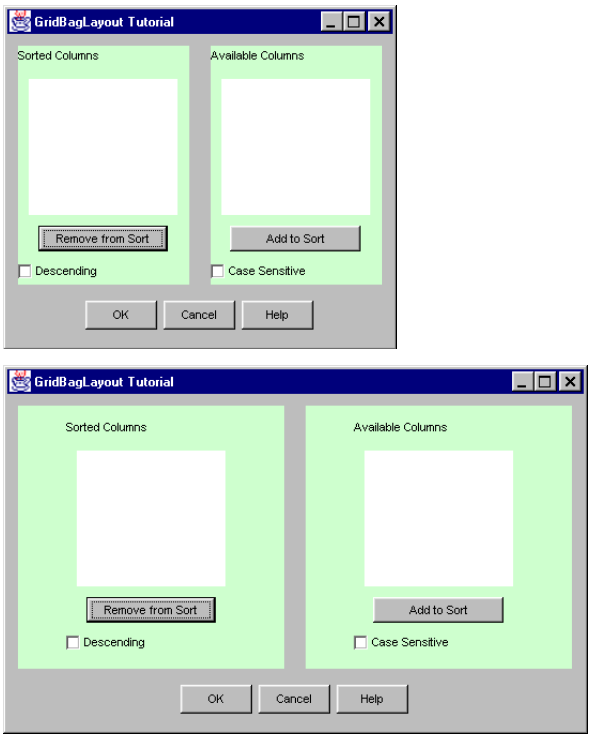
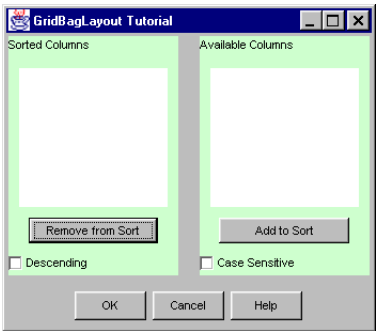


Figure 11.87 Weight constraints on the lists, but not on the panels



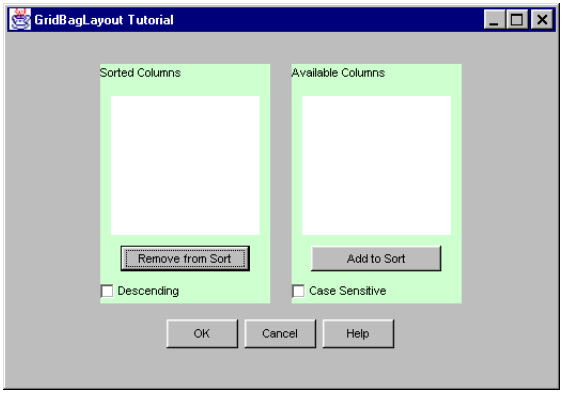


Figure 11.88 Horizontal weight constraints (weightx) only on all four components

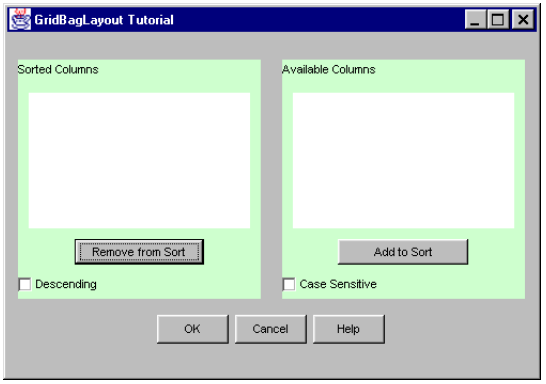


Figure 11.89 Vertical weight constraints (weighty) only on all four components

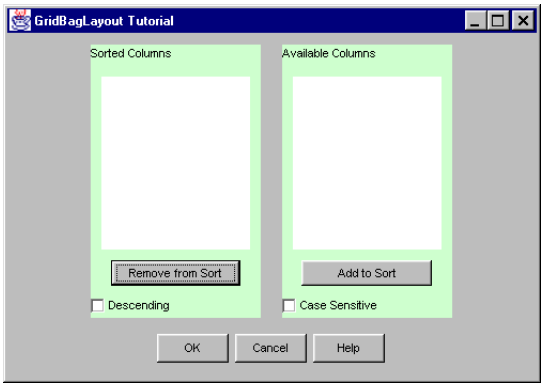
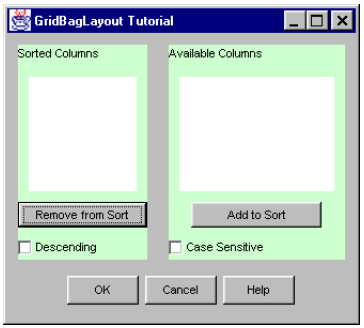
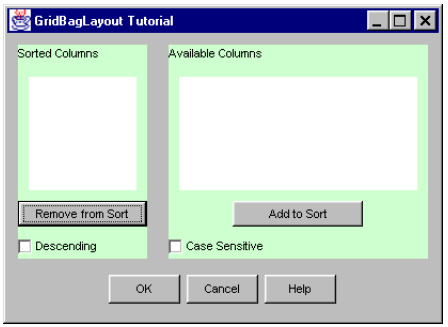


Figure 11.90 Weight constraints on only one panel and list component in the row





Appendix

A

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 1-6.

VisualAge

No modifications are necessary for VisualAge 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 in the Online Help to create a new JBuilder project that imports your existing source tree. The Project For Existing Code wizard is a feature of JBuilder SE 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

- The "Import VisualCafé Project wizard" topic in the 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

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X