Designing Applications: Tutorials

JBuilder® 2005

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Printed in the U.S.A.

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

Contents

Chapter 1

User interface tutorials

These tutorials are designed to help you quickly get comfortable with using the visual designer. Each tutorial has a different emphasis, so you can choose the tutorial that most closely matches your needs.

Here is a quick and simple tutorial to introduce you to the designer:

- Chapter 2, "Tutorial: Creating a simple user interface" — Create a simple GUI, play with layouts and constraints, and hook up an event.

These tutorials provide in-depth practice with important aspects of the designer:

- Chapter 3, "Tutorial: Building a Java text editor" — Create and deploy a real application to load, edit, and save text files, hooking events up to buttons and menus.
- Chapter 4, "Tutorial: Creating a UI with nested layouts" — Design a user interface with nested panels and layouts, converting between different layouts.
- Chapter 5, "Tutorial: Creating a GridBagLayout in JBuilder" — Learn how to use GridBagLayout and GridBagConstraints. Create a GridBagLayout UI container in the designer.

These tutorials assume you are familiar with Java and with the JBuilder IDE. For more information on these and other pertinent subjects, see below.

See also

- Java:

  Getting Started with Java
- JBuilder IDE:

  "The JBuilder environment" in *Getting Started with JBuilder*
- Using JBuilder features to improve ease of use for people with disabilities:

  Accessibility options section in the JBuilder Quick Tips

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
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."
`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`
[ ]	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. JDK 5.0 uses angle brackets to denote generics. 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. See "Using command-line tools" in *Building Applications with JBuilder* for more information. **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.
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, Linux, and OS X, 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 and XP, the home directory is `C:\Documents and Settings\<username>`
Screen shots	Screen shots reflect the Borland Look & Feel on various platforms.

Developer support and resources

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

Contacting Borland Developer Support

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

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

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

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

Online resources

You can get information from any of these online sources:

World Wide Web	`http://www.borland.com/` `http://info.borland.com/techpubs/jbuilder/`
Electronic newsletters	To subscribe to electronic newsletters, use the online form at: `http://www.borland.com/products/newsletters/index.html`

World Wide Web

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

You may want to check these URLs in particular:

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

Borland newsgroups

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

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

Usenet newsgroups

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

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

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

Reporting bugs

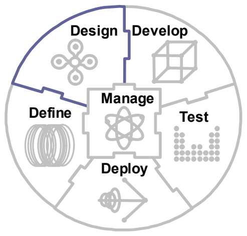
If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

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

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

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email `jpgpubs@borland.com`. 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.



Chapter

2

Tutorial: Creating a simple user interface

This is a very simple tutorial designed to acquaint you quickly with the designer. It takes ten or fifteen minutes to complete.

With a new project open (File|New Project),

1 Create a new application: use File|New, select the General page, and double-click Application.

2 Click Next in the Application wizard and make sure the checkboxes in the Options group are *not checked*. (You don't need these for this application.)

3 Click Finish in the Application wizard.

You now have an application with two JAVA files:

- `Application.java`, with the `main()` method which serves as the entry point into the program.
- `Frame1.java`, where the UI code goes.

Now you're ready to begin.

Creating the UI

1 Double-click `Frame1.java` in the project pane to open it in the content pane if it is not already open.

2 Click the Design tab at the bottom of the content pane. This opens the UI designer.

Notice that only one component is in `this`: the `contentPane`. The `contentPane`'s default layout is BorderLayout, meaning that components dropped into this container will cling to the North, South, East, or West margin, or will fill the Center. We want to design the UI before determining which layout to use, so we will change the layout property to one that will allow us to manipulate the design freely.

1 Select `contentPane` in the component tree (which appears in the structure pane.)

2 Look at the Inspector, on the right of the design surface. Look for the `layout` property.

Tip Properties and events are listed alphabetically in their tables in the Inspector.

3 Click in the right-hand column, which says `BorderLayout` for the layout property.

   A list of the layout managers appears.

4 Select `null`.

Note `null` layout imposes no layout behavior of its own. When you place a component on a container with null layout management, the component goes exactly where you put it as you resize and move it on the design surface. Once the design is finished, you change the layout to a layout manager that is responsive. This way, your UI can be resized dynamically at runtime, and can be ported to any platform. You can still make modifications to the layout in the designer after converting to the intended layout.

Now let's add a component to our null layout `contentPane`:

1 Look at the component palette along the left edge of the design surface. It contains many pages of components which can be expanded or collapsed. The Swing page is expanded by default.

2 Click the JButton icon on the Swing page. This loads a `javax.swing.JButton` component into your cursor.

3 Click in the contentPane on the design surface, about where you want the button's top left corner to be. The button drops into the contentPane.

4 Since we're using null layout, we can move the button wherever we like. Let's put it in the lower right corner, not too close to the edges.

`jButton1` is the default name. Let's change it to something more meaningfull:

1 Right-click `jButton1` in the component tree and choose Rename.

2 Clear the old name, type in `pushButton`, and press *Enter*.

   This command refactors the name `jButton1` to `pushButton` wherever this particular JButton component is referred to.

Using layouts

Now, let's apply a portable layout.

1 Select `contentPane` in the component tree.

   This makes the Inspector display all the properties for the parent container of the button.

2 Look at the Inspector. Click the value `null` next to `layout`.

   The other layouts become available.

   Take a good look at where the button is. Most layouts are going to have trouble with this, because it's one single component in an off-center position in the container. However, GridBagLayout can handle this. (With more complex UIs, this layout manager is challenging to master, but it's sufficiently powerful and flexible that it's worth the effort.)

3 Select GridBagLayout from the list of layouts.

4 Run your application.

If you like, you can play with the application. Notice that, when you resize it small, the button doesn't stay inside the viewable part of the container. That's because JBuilder's Constraints editor for GridBagLayout has chosen constraints that most closely represent your base design. These constraints are strict, but they can be modified.

Adjusting constraints

1 Click on the button in the designer, and in the Inspector, look at the `constraints` property.

2 Click the cell that lists the constraints. A small ellipsis (…) button appears on the right edge of the cell.

3 Click the ellipsis (…) button.

   The GridBagConstraints editor appears.

4 Notice how many values are filled in, and with what high numbers. The Constraints editor was trying very hard to translate the null layout design as precisely as it could. We want it to be more flexible.

5 Start with the Anchor area. Center is currently selected. Choose SE instead.

6 Next, look at External Insets. These determine how much space is around a component.

   Make the values for Top and Left `0` (zero). This is because, since the component is anchored at SouthEast, it doesn't need to be pushed down or over any further.

7 Make the values for Bottom and Right `30`. This provides a nice amount of padding, keeping the button comfortably off the lower right edges.

8 Click the Apply button in the editor. This shows you what it will look like in the designer.

9 Once you're happy with the layout, click OK in the Constraints editor.

10 Run your application and resize it. The button stays at the same comfortable distance from the bottom right corner, and moves sensibly when the container is resized.

Attaching events

Next, we'll hook an action up to the button.

1 In the component tree, select `pushButton`.

   We normally select components in the component tree in order to avoid confusion when our UI gets more complex. You can also select components by clicking on them in the design surface.

Tip When you're in the design surface, look in the status bar at the bottom left to determine exactly which component your cursor is over.

2 In the Inspector, look at the `text` property. It says `jButton1`. Change it to `Push Me`.

3 In the Inspector, click the Events tab.

   This gives us access to all the events available to that component.

4 Look for the `actionPerformed()` event, at or near the top. Double-click the right-hand cell next to it.

   This does two things: it creates a stub of the method in the code of `Frame1.java`, and it switches you to the editor with your cursor right inside the stub, where you enter the code that will define the action.

5 Type this into the stub:

```
pushButton.setText("I've been pushed!");
```

Your method now looks like this:

```
public void pushButton_actionPerformed(ActionEvent e) {
    pushButton.setText("I've been pushed!");
}
```

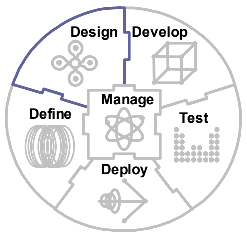
6 Save and run the program, then push the button.

It works!

You've been introduced to the major parts of the designer. In addition, you've been introduced to basic strategies useful throughout the designer and the rest of the IDE:

- Using the object gallery to access wizards.
- Using keyboard shortcuts and right-click context menus to access commands quickly.
- Using single-click, double-click, and right-click in different areas.
- Using the different panes in the IDE in a coordinated manner.

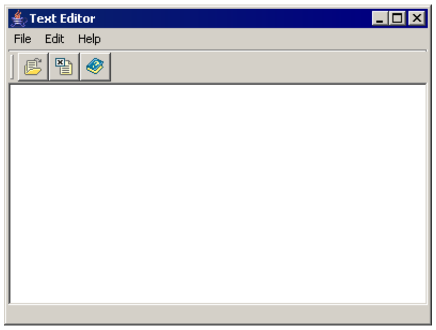
More tutorials are available for hooking up events (Chapter 3, "Tutorial: Building a Java text editor"), using layouts (Chapter 4, "Tutorial: Creating a UI with nested layouts"), and using GridBagLayout (Chapter 5, "Tutorial: Creating a GridBagLayout in JBuilder").



Chapter
3

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 font, color, and background color of the text editing region.

The tutorial takes approximately two hours to complete.

What this tutorial demonstrates

Some steps in this tutorial are specific to JBuilder Developer 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.
- 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|Save As, editing, and exiting the file.
- Deploy the “Text Editor” application to a JAR file. This is a feature of JBuilder Developer 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.

This tutorial assumes you are familiar with Java and with the JBuilder IDE. For more information on Java, see *Getting Started with Java* For more information on the JBuilder IDE, see “JBuilder’s work environment” in *Getting Started with JBuilder.*

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see “Documentation conventions” on page 2.

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 Developer and Enterprise, or to look at the FontChooser code:

```
<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

- “Visual design in JBuilder” in *Designing Applications with JBuilder*
- “Using the JBuilder workspace” in *Getting Started with 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 Select the Java Formatting|Generated page in the Project Properties dialog box.

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

4 Look under Event Handling.

5 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.

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

7 Accept the default Package value for Instance Variable Visibility.

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.

9 Click OK to close the Project Properties dialog.

See also

- "Choosing event handler style" in *Designing Applications with JBuilder*

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|General.

2 Double-click the Application icon to open the Application wizard.

3 Change the application class name in Step 1:

- Class name: `TextEditClass`

Accept the default package name.

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

5 Change the name and title of the frame class:

- Class: `TextEditFrame`
- Title: `Text Editor`

6 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.

7 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.

8 Click the Finish button.

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

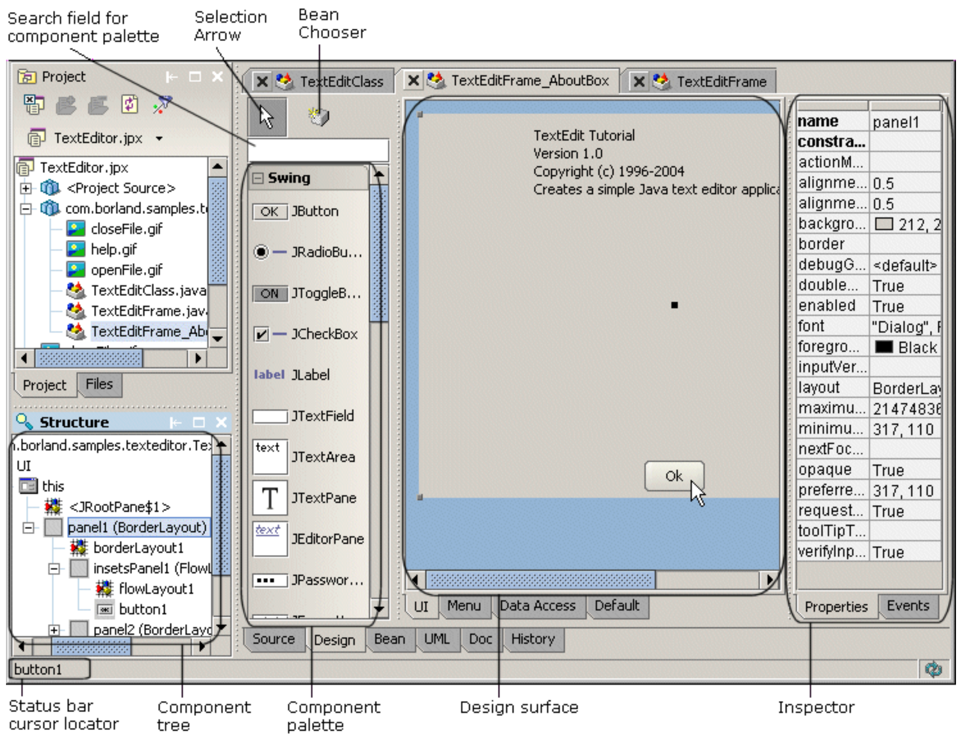
Note An automatic source package node also appears in the project pane if the Automatic Source Packages option is enabled on the General page of the Project Properties dialog box (Project|Project Properties).

9 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 content pane, 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 3.1 JBuilder in design view



TextEdit Tutorial
Version 1.0
Copyright (c) 1996-2004
Creates a simple Java text editor applica

name	panel1
constra...	
actionM...	
alignme...	0.5
alignme...	0.5
backgro...	212, 2
border	
debugG...	<default>
double...	True
enabled	True
font	"Dialog",
foregro...	Black
inputVer...	
layout	BorderLa
maximu...	21474836
minimu...	317, 110
nextFoc...	
opaque	True
preferre...	317, 110
request...	True
toolTipT...	
verifyInp...	True

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 workspace, adjust the size of the 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.v

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

You can set the look and feel for the designer 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" in *Designing Applications with JBuilder*

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" in *Designing Applications with JBuilder*

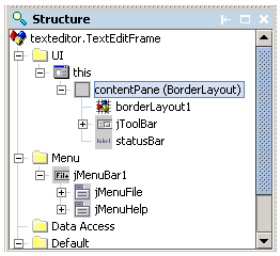
The Application wizard creates a `JFrame` component that's the main container for this UI. This `JFrame` component is the `this` component. `this` contains a `JPanel` object called `contentPane` which already uses `BorderLayout`. All you need to do now is add the components for the text area to the `contentPane`.

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.

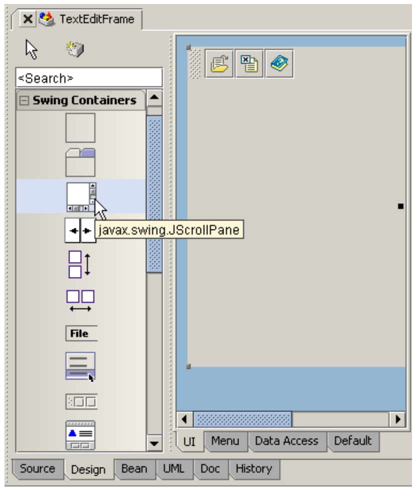
1 Select the `TextEditFrame` tab in the content pane.

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 page on the component palette and select the `JScrollPane` component.



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

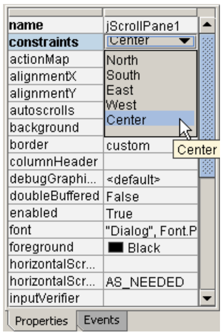
This drops the `JScrollPane` component into the `contentPane` panel and should give it a constraint of Center.

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.



name	jScrollPane1
constraints	Center
actionMap	
alignmentX	
alignmentY	
autoscrolls	
background	
border	custom
columnHeader	
debugGraphi...	<default>
doubleBuffered	False
enabled	True
font	"Dialog", Font.P
foreground	■ Black
horizontalScr...	
horizontalScr...	AS_NEEDED
inputVerifier	

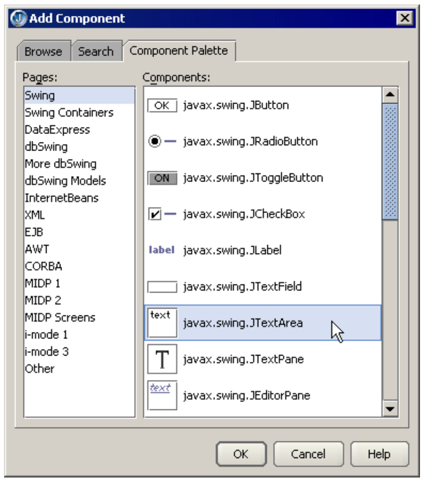
Let's add the text area, using a different way of adding components to the design. We'll use the Add Component dialog box.

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:



Pages:
Swing
Swing Containers
DataExpress
dbSwing
More dbSwing
dbSwing Models
InternetBeans
XML
EJB
AWT
CORBA
MIDP 1
MIDP 2
MIDP Screens
i-mode 1
i-mode 3
Other

Components:
javax.swing.JButton
javax.swing.JRadioButton
javax.swing.JToggleButton
javax.swing.JCheckBox
javax.swing.JLabel
javax.swing.JTextField
javax.swing.JTextArea
javax.swing.JTextPane
javax.swing.JEditorPane

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`.

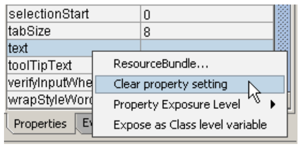
When you paste the text area component back in, this will be its parent.

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 Select jTextArea1 in the component tree and right-click the `text` property in the Inspector. 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

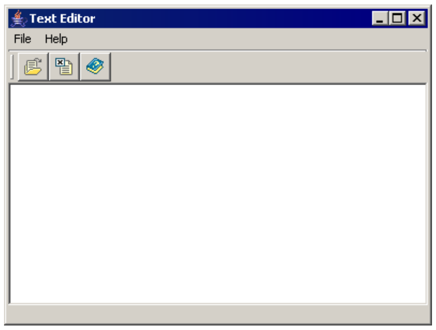
Now, compile your program then run it to see how it looks.

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.

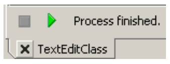
Your runtime UI should now look like this:



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.

3 Choose File|Exit in the "Text Editor" application to close the runtime window.

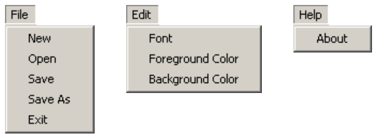
4 To close the message pane, click the close button in the corner of the message tab:



Next, we will create and populate useful menus.

Step 3: Creating the menus

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



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: Expand the items in the Menu folder in the component tree and either double-click `jMenuFileExit`, or select it and click the Menu tab in the content pane.

   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 before the Exit item. 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

  b Foreground Color

  c Background Color

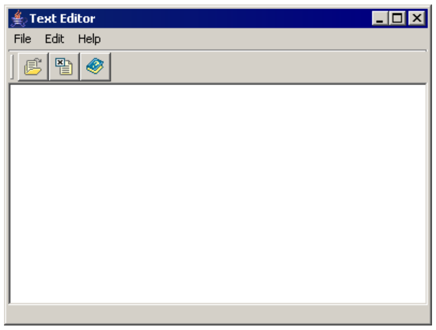
  If an entry is too wide for the edit area, the text automatically scrolls as you type. When you press *Enter*, the Menu designer adjusts the width of the menu to accommodate the longest item in the list.

5 Close the Menu designer by clicking the UI tab in the content pane or by double-clicking UI folder in the component tree.

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

6 Save the file, then compile and 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" in *Designing Applications with JBuilder*

Step 4: Adding a 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. Expand the More dbSwing page on the palette and select the `com.borland.dbswing.FontChooser` component.
3. Click anywhere in the designer, or click on the `Default` folder in the component tree. The component `fontChooser1` will be added to the `Default` folder.

Note 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 the `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

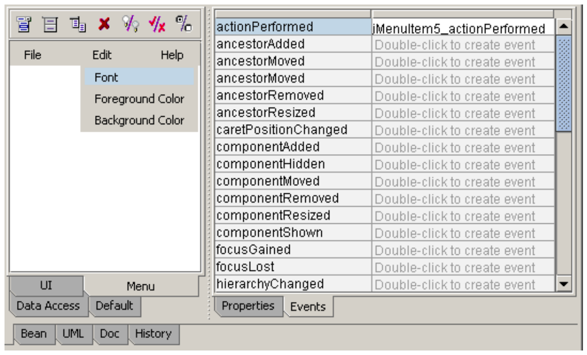
Create an event for the Edit|Font menu item that will launch the `FontChooser`:

1. Click the Menu tab to switch back to the Menu designer and select the Edit|Font menu item.
2. Click the Events tab in the Inspector.

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

3 Double-click the value field (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.



actionPerformed	jMenuItem5_actionPerformed
ancestorAdded	Double-click to create event
ancestorMoved	Double-click to create event
ancestorMoved	Double-click to create event
ancestorRemoved	Double-click to create event
ancestorResized	Double-click to create event
caretPositionChanged	Double-click to create event
componentAdded	Double-click to create event
componentHidden	Double-click to create event
componentMoved	Double-click to create event
componentRemoved	Double-click to create event
componentResized	Double-click to create event
componentShown	Double-click to create event
focusGained	Double-click to create event
focusLost	Double-click to create event
hierarchyChanged	Double-click to create event

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:

```
public 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" runtime window and 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

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 (`public void jMenuItem5_actionPerformed(ActionEvent e)`) that you just created, if you're not there already.

2 Insert the following code into your Font menu item (`jMenuItem5`) event handling method between the opening and closing curly braces, replacing the existing `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:

```
public 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.

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" runtime window and 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 Foreground Color menu item (`jMenuItem6`).
3. Click the Events tab in the Inspector and double-click the `actionPerformed()` event to create the following event handler:

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

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

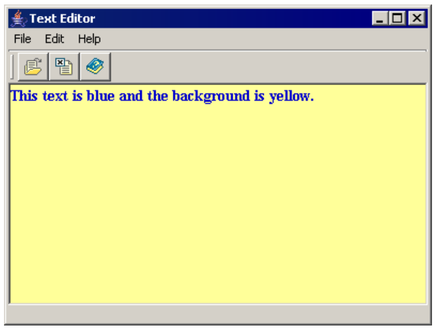
```
// Handle the "Foreground Color" menu item
Color color = javax.swing.JColorChooser.showDialog(this,"Foreground Color",
    jTextArea1.getForeground());
if (color != null) {
  jTextArea1.setForeground(color);
}
```

5. Click the Design tab (the Menu designer should still be open.)
6. Select the Background Color menu item (`jMenuItem7`) and 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 = javax.swing.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. If it works, you'll see the changes. We set ours to blue foreground, yellow background.



This text is blue and the background is yellow.

9 Close the "Text Editor" runtime window and the JBuilder message pane.

Next, we'll activate the File|New item.

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 and select the File|New menu item (`jMenuItem1`.)

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

3 Insert the following code into it:

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

4 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.

5 Close the "Text Editor" runtime window and 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 on 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 (`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 Step 9.

7 Close the "Text Editor" runtime window before continuing.

Internationalizing Swing components

JBuilder Personal users skip this step.

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. 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 Find the File|Open event handler in the source code (`jMenuItem2_actionPerformed(ActionEvent e)`).

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

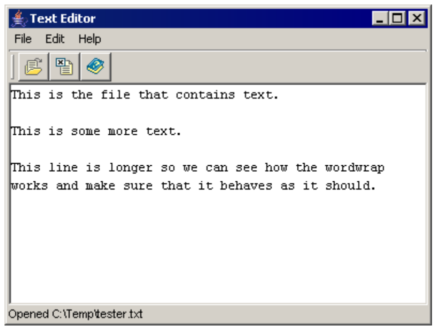
```
// Display the name of the opened directory+file in the statusBar.
statusBar.setText("Opened "+jFileChooser1.getSelectedFile().getPath());

// 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:



```
This is the file that contains text.

This is some more text.

This line is longer so we can see how the wordwrap
works and make sure that it behaves as it should.
```

Opened C:\Temp\tester.txt

7 Close the "Text Editor" runtime window and the JBuilder message pane.

We'll continue hooking menu items to menu events. Next, we'll make it so that the application can save changes to a file.

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 (`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" runtime window and 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 Find the File|New event-handling method in the code:

```
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:

```
public 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:

```
public 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" runtime window and 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:

```
public 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” runtime window and 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.

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

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

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" runtime window and the JBuilder message pane.

Step 14: Adding a context menu to the text area

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.

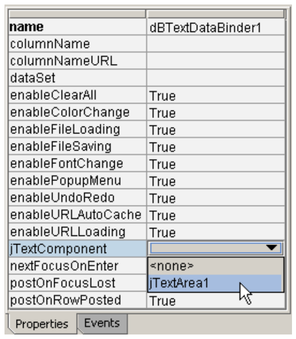
`DBTextDataBinder` also has built-in actions to load and save files into a `JTextArea`, but they don't allow you to retrieve the file name loaded or saved. Unfortunately, we want this information because this application displays the file name in the status bar. Therefore, we are going to add a `DBTextDataBinder`,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



name	dBTextDataBinder1
columnName	
columnNameURL	
dataSet	
enableClearAll	True
enableColorChange	True
enableFileLoading	True
enableFileSaving	True
enableFontChange	True
enablePopupMenu	True
enableUndoRedo	True
enableURLAutoCache	True
enableURLLoading	True
jTextComponent	
nextFocusOnEnter	<none>
postOnFocusLost	jTextArea1
postOnRowPosted	True

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`

8 Close the "Text Editor" runtime window and the JBuilder message pane.

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.

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

In this 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 Locate the `jMenuFileExit_actionPerformed(ActionEvent e)` event handling method in the code.

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(ActionEvent e)`:

```
public 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:

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

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

public 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|Save As.
- 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.
- 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`. JBuilder Developer and Enterprise users, Proceed to Step 16 to deploy this application and run it from the command line.

To comment on or make suggestions for this tutorial, please send email to `jpgpubs@borland.com`.

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

This step is for JBuilder Developer 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– 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 particularly. 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 File|New and select the Archive page. Double-click the Application archive icon.

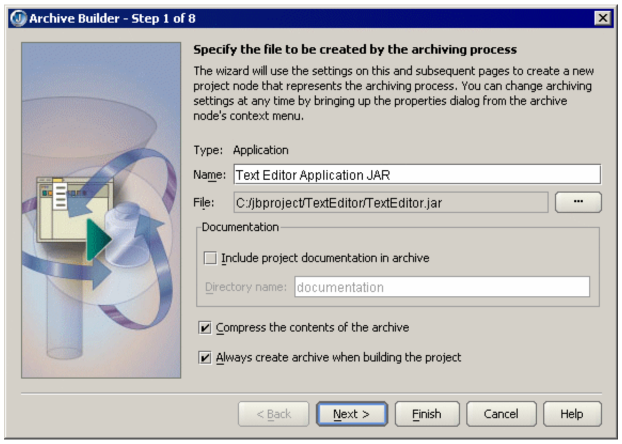
Step 1 of the Archive Builder appears.

3 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.

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

5 Accept the remaining defaults on this page.

When you're done, Step 1 of the wizard should look like this:



Specify the file to be created by the archiving process

The wizard will use the settings on this and subsequent pages to create a new project node that represents the archiving process. You can change archiving settings at any time by bringing up the properties dialog from the archive node's context menu.

Type: Application

Name: Text Editor Application JAR

File: C:/jbproject/TextEditor/TextEditor.jar

Documentation

[ ] Include project documentation in archive

Directory name: documentation

[✓] Compress the contents of the archive

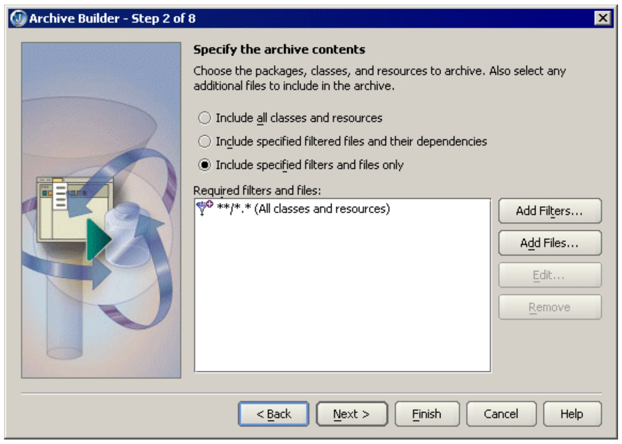
[✓] Always create archive when building the project

6 Click Next to go to Step 2 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 applying filters, thereby excluding unnecessary files and classes, or adding classes and files. Use the Add Filters and Add Files buttons to tune the archive file. Use the Edit and Remove buttons to change or delete existing filters and included files. Then test the deployed application to be sure you've included all the necessary files.

Step 2 of the wizard will look like this:


Archive Builder - Step 2 of 8

Specify the archive contents

Choose the packages, classes, and resources to archive. Also select any additional files to include in the archive.

- ○ Include all classes and resources
- ○ Include specified filtered files and their dependencies
- ● Include specified filters and files only

Required filters and files:

**/*.* (All classes and resources)

See also

- "Adding filters" in *Building Applications with JBuilder* to learn how to use the filters on this page.

7 Click Next to go to Step 3 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.

Note You set archive contents separately for each item in the list.

a Select dbSwing Dependency Rule column.

b Choose the Dependencies and Resources option from the drop-down list.

c Select DataExpress from the list.

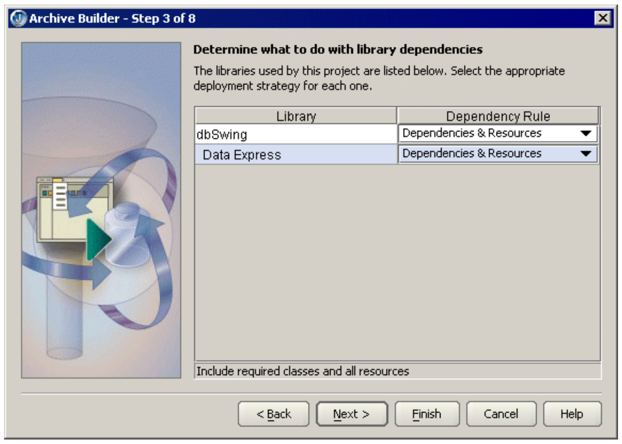
d Choose Dependencies and Resources.

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.

Both libraries are deployed with Dependencies and 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 3 of the wizard should look like this:



Determine what to do with library dependencies

The libraries used by this project are listed below. Select the appropriate deployment strategy for each one.

Library	Dependency Rule
dbSwing	Dependencies & Resources
Data Express	Dependencies & Resources

Include required classes and all resources

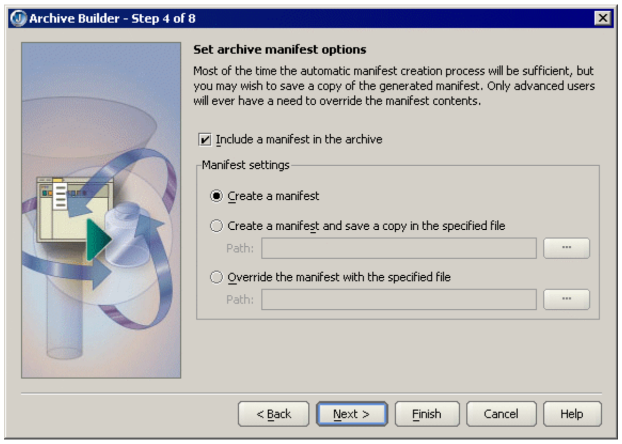
8 Click Next to go to Step 4, where you create the manifest file.

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

9 Accept the default settings for Step 4 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 4 of the wizard will look like this:



Set archive manifest options

Most of the time the automatic manifest creation process will be sufficient, but you may wish to save a copy of the generated manifest. Only advanced users will ever have a need to override the manifest contents.

✓ Include a manifest in the archive

Manifest settings

● Create a manifest

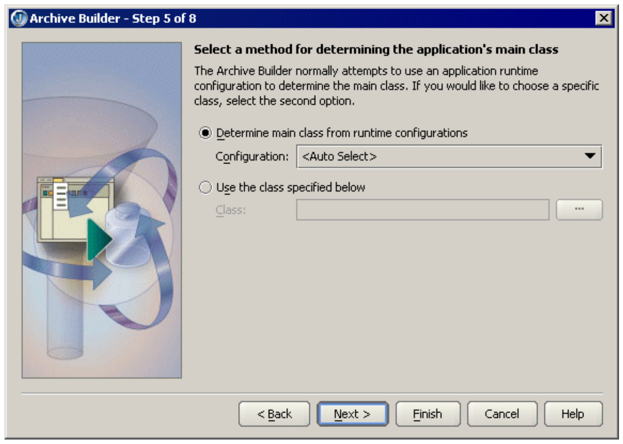
○ Create a manifest and save a copy in the specified file

○ Override the manifest with the specified file

10 Click Next to go to Step 5, 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 5 of the wizard will look like this:



Select a method for determining the application's main class

The Archive Builder normally attempts to use an application runtime configuration to determine the main class. If you would like to choose a specific class, select the second option.

(●) Determine main class from runtime configurations

Configuration: <Auto Select>

( ) Use the class specified below

Class:

11 Click Finish to create the archive node. There are additional steps to the wizard, but you accept the default settings for those when you press Finish.

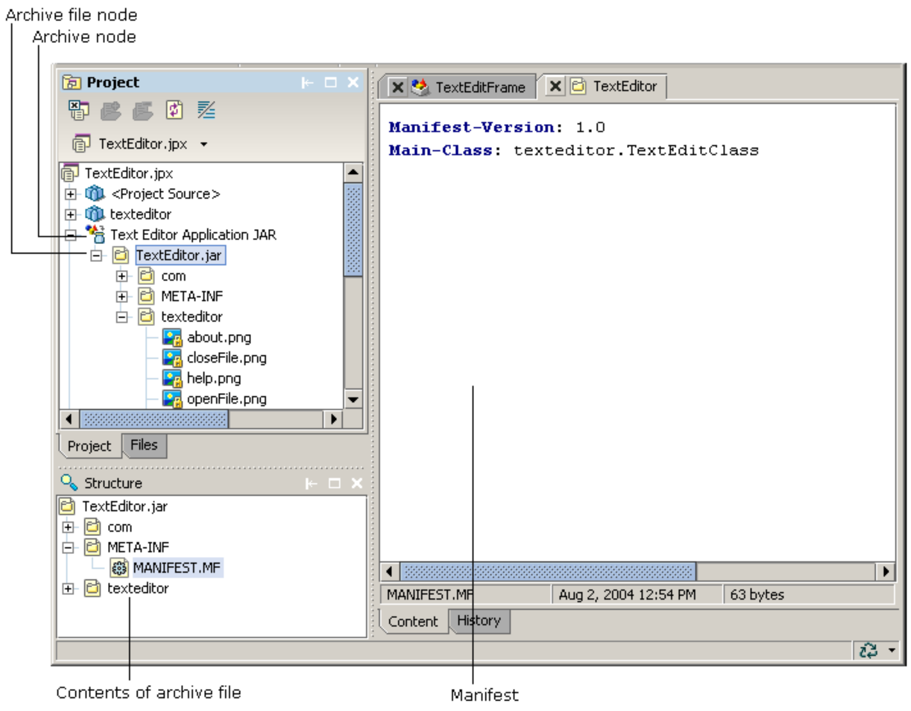
The archive node, `Text Editor Application JAR`, is now displayed in the project pane. You can right-click the archive node and make it, rebuild it, or change its properties.

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

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

3 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:



```
Manifest-Version: 1.0
Main-Class: texteditor.TextEditClass
```

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

Some developers like to test their applications outside JBuilder. Before you run your 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 46 and test the application when it runs.

That's it!

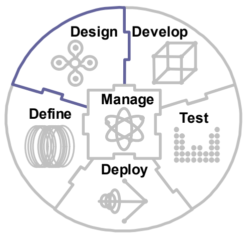
As you can see, there is a lot of information to assimilate related to deployment. Deployment goes far beyond just creating an archive file. Not only do you have to make sure you provide all the necessary classes, resources, and libraries in your deployment set, you have to be concerned with other issues, 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*
- "Using the Archive Builder" 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`.

To suggest ways to improve this tutorial, send email to `jpgpubs@borland.com`.



Chapter
4

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 "Using layout managers" in *Designing Applications with JBuilder* 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 page of the component palette at the left of JBuilder's UI designer. Throughout this tutorial, any reference to a panel implies `JPanel`.

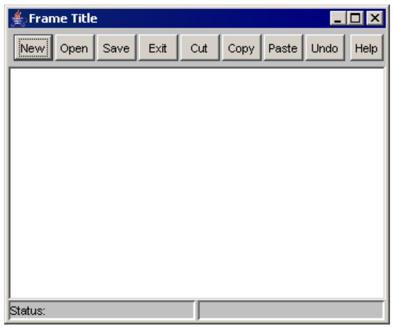
Also in this tutorial, you'll change some of the panel layout managers during the design phase to `XYLayout` or `null` layout. `XYLayout`, a JBuilder custom layout manager, places components in a container at specific x,y coordinates relative to the upper left corner of the container. Regardless of the 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 "Creating a GridBagLayout in JBuilder" on page 76. 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:



This tutorial assumes you are familiar with Java and with the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on the JBuilder IDE, see "The JBuilder environment" in *Getting Started with JBuilder.*

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 2.

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 *Building Applications with JBuilder* for more information on step 2 of the Project wizard.
- "Where are my files?" in *Building Applications with JBuilder*

Next, you'll create the application files using the Application wizard.

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

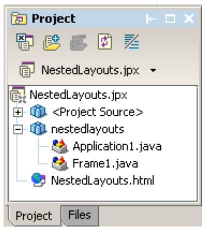
To open the Application wizard,

1 Choose File|New|General.

2 Double-click the Application icon to open the Application wizard. Note the default package name which is extracted from the project name.

3 Click Finish. We can safely do this because 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.

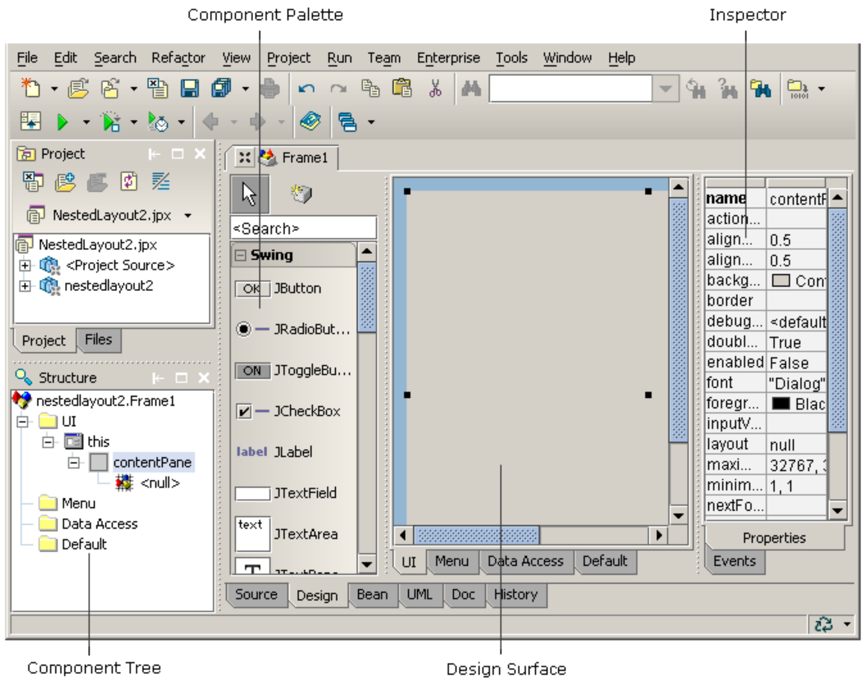
The project appears in the project pane of the workspace with `Frame1.java` open in the content pane. You'll see two additional files in the project pane: `Application1.java`, which contains the `main()` method, and `Frame1.java`, the UI container.



Note 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.

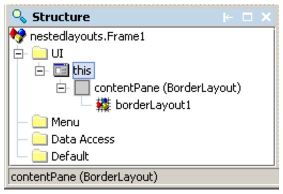
Figure 4.1 UI designer



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 on the left.

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.



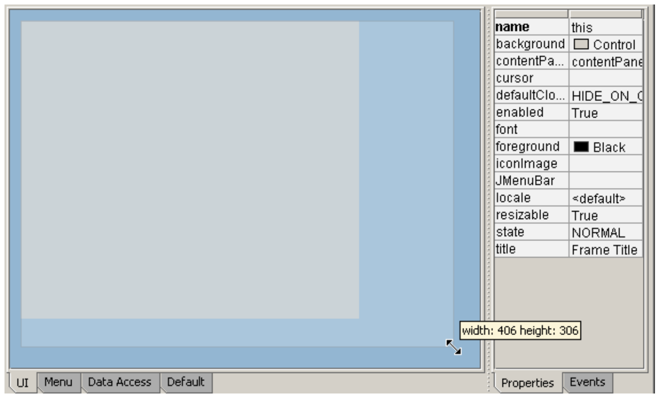
2 Look at the Properties page of the Inspector and find the value of the `title` property, which is "Frame Title".

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 JBuilder before you drag the frame to a larger size. If you want even more space, hide the project and structure panes by selecting View|Panes and uncheck the panes you want to hide. You can also make the designer wider by dragging the splitter bars on either side.



name	this
background	▭ Control
contentPa...	contentPane
cursor	
defaultClo...	HIDE_ON_(
enabled	True
font	
foreground	■ Black
iconImage	
JMenuBar	
locale	<default>
resizable	True
state	NORMAL
title	Frame Title

width: 406 height: 306

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.

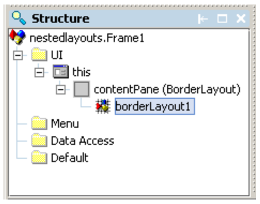
You can manually change the size of the UI frame by resizing it at runtime.

In the next step, you'll change the layout manager of the `contentPane`.

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.



```
Structure
nestedlayouts.Frame1
⊟ UI
  ⊟ this
    ⊟ contentPane (BorderLayout)
        borderLayout1
  Menu
  Data Access
  Default
```

`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.

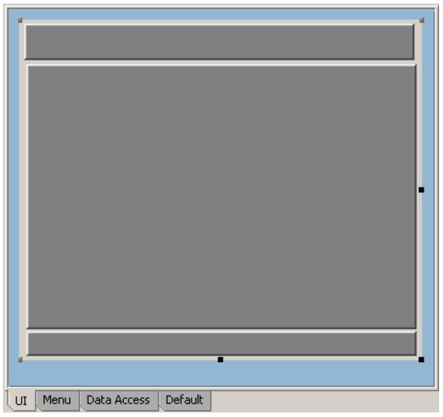
3 Click the drop-down arrow and select `XYLayout` or `null` from the list.

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.

Next, begin adding panels to the `contentPane`.

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:



Each of these three panels 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.

1 Select `contentPane` in the component tree.

2 Expand the Swing Containers page on the component palette 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 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 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.

6 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.

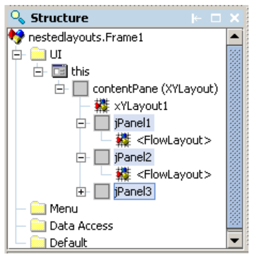
4 Right-click with the mouse over one of the selected panels and choose Same Size Horizontal. The middle and bottom panels will snap to the same width as the top panel. Then select Align Left and Even Space Vertical from the designer's context menu.

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:



```
Structure
nestedlayouts.Frame1
  UI
    this
      contentPane (XYLayout)
        xYLayout1
        jPanel1
          <FlowLayout>
        jPanel2
          <FlowLayout>
        jPanel3
  Menu
  Data Access
  Default
```

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

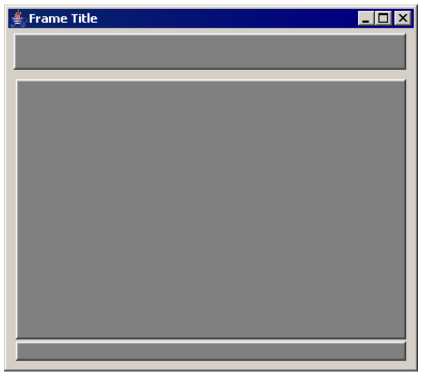
1 Select each panel in the component tree. Right-click the panel and select Rename from the menu. Give each panel the appropriate name:

- 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.

In the next step, you'll add additional panels to the top panel.

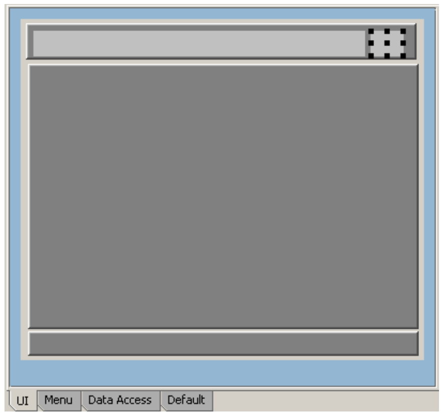
Step 5: Creating toolbars

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.

In the next step, you'll add buttons to the toolbars.

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 page 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`

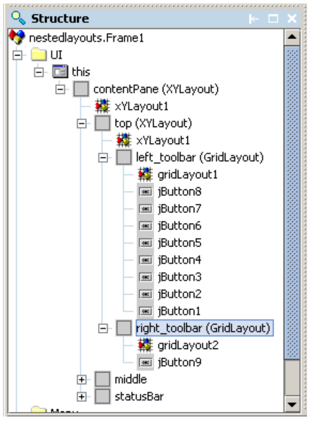
Important You may not be able to see the text on the buttons yet because the margins need to be adjusted. You'll do that to all the buttons after you finish adding them.

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 Add one button (`jButton9`) to 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.

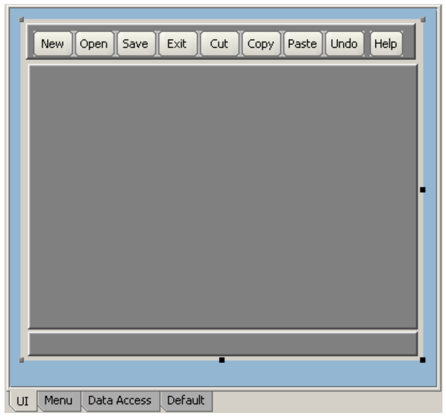
2 Change the `margin` property to `2,2,2,2`.

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. Notice the space added between the buttons.

Your design should now look something like this:



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

Next, you'll add components to the middle panel.

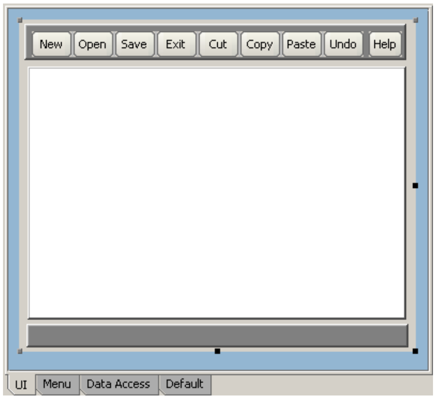
Step 7: Adding components to the middle panel

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 page on the component palette.
3 Select the `JScrollPane` component and drop onto the middle panel in the designer or the component tree. `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 page 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*.

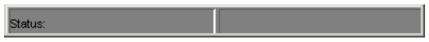
Your UI should look similar to this now:



Next, you'll create the status bar.

Step 8: Creating a status bar

Although the status bar 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.

label

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.

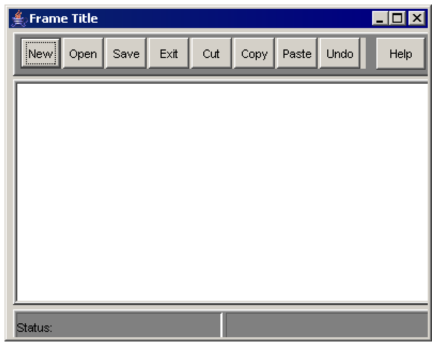
Important In JBuilder, you cannot edit the `layout` properties for a `<default layout>`. If you want to modify the properties for a container's layout manager, you must specify a specific layout manager. Then its properties are accessible in the Inspector.

5 Select the left label and change the default value in the left label's `text` property to `Status:`. Also set it's `horizontalAlignment` property to LEFT.

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 the one below.



Notes 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");
```

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.

Next you'll start converting the panels to other layouts.

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.

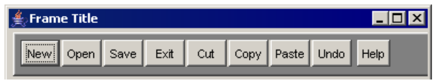
Now, change the layout for the top panel that contains the toolbars and align the toolbars to the left.

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.

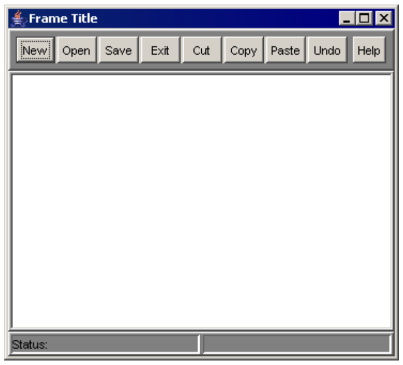
5 Select the top panel in the designer and make it narrower than the buttons to see what `FlowLayout` does when the panel is narrower. Notice that the Help button on the `right_toolbar` panel moves to the second row. This is `FlowLayout` behavior. Adjust the panel until both toolbars are back on the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

  Finally, change the layout for `contentPane` to `BorderLayout`.

6 Select `contentPane` in the component tree and change the layout property 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.

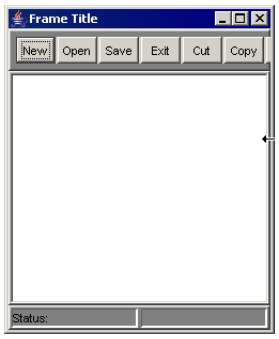
7 Save your work and run the application.

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.

Next, move on to the last step, where you'll fine-tune and finalize your UI.

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.

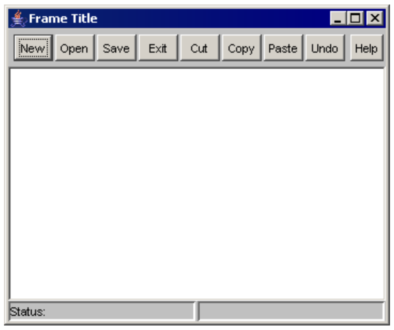
1. Select all the panels in your design using multiple selection in the component tree.
2. Double-click the `background` property in the Inspector and select light gray (RGB value 192, 192, 192). Don't worry about its appearance in the designer at this point. The bevels create a lot of white space. You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



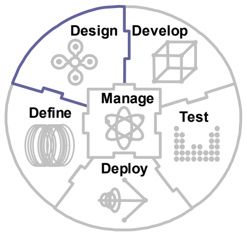
And that's all there is to it! While it seems slow in the learning phase, once you become familiar with the different layouts, you'll be able to plan and implement layouts more quickly.

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 5, "Tutorial: Creating a GridBagLayout in JBuilder."

To learn how to write code that responds to user events in your application, see Chapter 3, "Tutorial: Building a Java text editor."

This concludes the tutorial.



Chapter

5

Tutorial: Creating a GridBagLayout in JBuilder

This tutorial 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 tutorial is valid for all supported platforms, as the functionality of JBuilder and `GridBagLayout` is the same.

This tutorial uses `XYLayout` for prototyping the UI. If you prefer, substitute `null` layout wherever `XYLayout` is mentioned.

This tutorial assumes you are familiar with Java and with the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on the JBuilder IDE, see “JBuilder’s work environment” in *Getting Started with JBuilder.*

This tutorial also assumes that you are at least familiar with `GridBagLayout`, although the tutorial can be completed without necessarily understanding how it works beforehand.

See also

- “GridBagLayout” in *Designing Applications with JBuilder* for an introduction to GridBagLayout concepts.
- “Using GridBagLayout” on page 66 for a thorough exploration of GridBagLayout and GridBagConstraints.
- “Accessibility options” in the JBuilder Quick Tips contains tips on using JBuilder for people with disabilities.
- “Documentation conventions” on page 2 for information on documentation conventions used in this tutorial and other JBuilder documentation.

The “GridBagLayout tutorial” is divided into three sections:

- “Using GridBagLayout” on page 66

  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.

- “Creating a GridBagLayout in JBuilder” on page 76

  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.

- “Tips and techniques” on page 94

  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 113

    An overview of the `GridBagConstraints` and their values.

  - “Examples of weight constraints” on page 116

    Illustrated examples of weight constraints applied in different ways.

Using GridBagLayout

`GridBagLayout` is a complex layout manager that requires some study and practice to understand. 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, take advantage of 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” in *Designing Applications with JBuilder*.

This tutorial uses `XYLayout` which is a proprietary JBuilder layout. If you prefer, substitute `null` layout for `XYLayout` wherever it’s mentioned.

What is GridBagLayout?

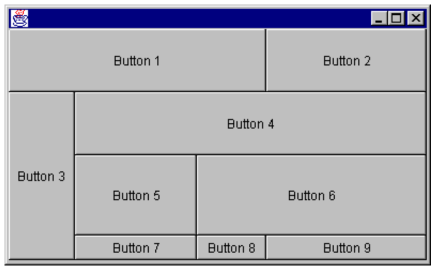
In Java, you create a user interface by adding components to a `Container` object, such as `Frame` or a `Panel`, and using a layout manager to control the size and placement of the objects within the container. By default, every `Container` object has a layout manager object that controls the layout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

`GridBagLayout` is an extremely flexible and powerful layout manager that implements the interface LayoutManager2. `GridBagLayout` determines where and how to place the components in a container based on the components' GridBagConstraints. It arranges 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 5.1 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?".

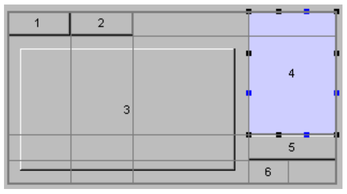
What is the component's display area?

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.

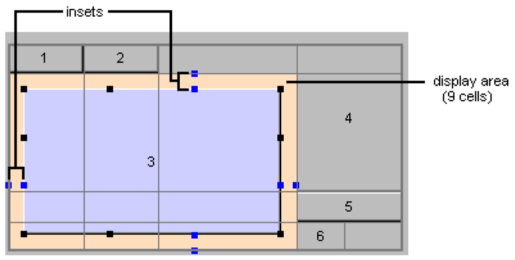
Figure 5.2 Display area — one horizontal cell, two vertical cells



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 5.3 Display area — three horizontal cells, three vertical cells

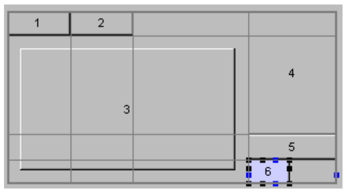


Even though this component has both horizontal and vertical `fill` constraints, since it also has `insets` on all four sides of the component (represented by the double blue nibs on each side of the display area), these take precedence over the `fill` constraints. The result is that the component only 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 5.4 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 later in "Tips and techniques" on page 94.

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

- `http://java.sun.com/j2se/1.3/docs/api/java/awt/GridBagConstraints.html`
- `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 causing the bottom row to expand.

Figure 5.5 Area 4 weighty constraint value is 1.0

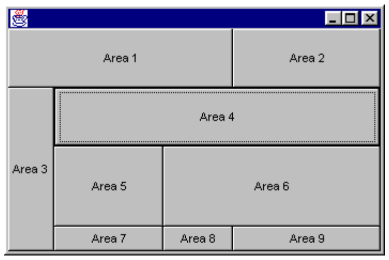
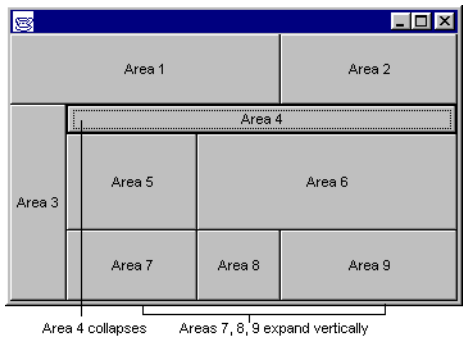


Figure 5.6 Area 4 weighty constraint value changed to 0.0



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 make.

Whether you are new to Java or you 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 designer
- Prototype your design in null or 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 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 "Creating a GridBagLayout in JBuilder" on page 76.

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 null or XYLayout" on page 75.

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 how `GridBagLayout` has difficulty determining 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 5.7 Drawing the new button in the middle

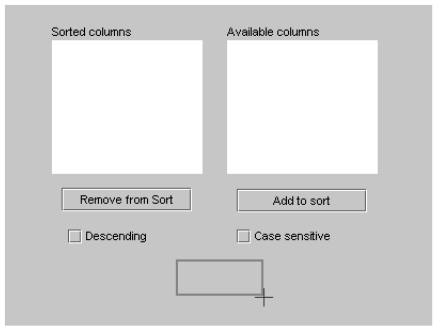
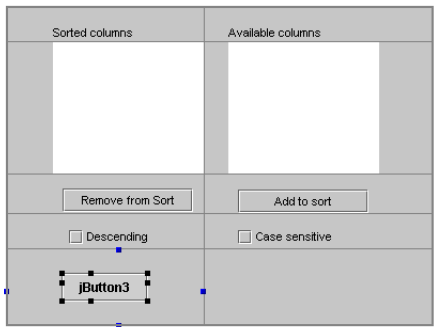


Figure 5.8 GridBagLayout snaps the button to column 1 when it is dropped



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 5.9 Dragging the new button back to the middle

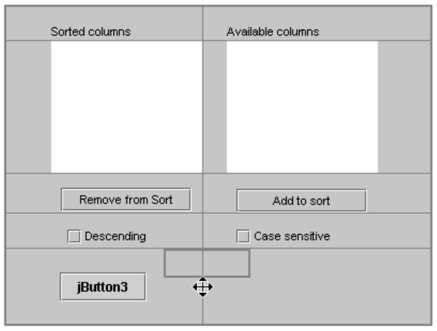
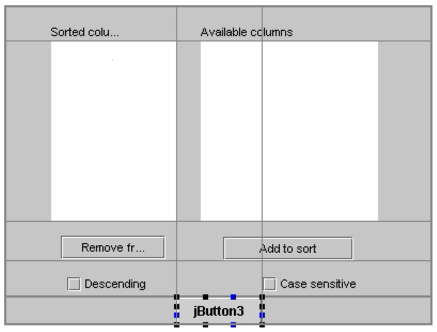


Figure 5.10 GridBagLayout creates a new center column for the button



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 to the container while it was still in `XYLayout` would have enabled us to place it exactly where and how we wanted it 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 null or XYLayout" on page 75.

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.

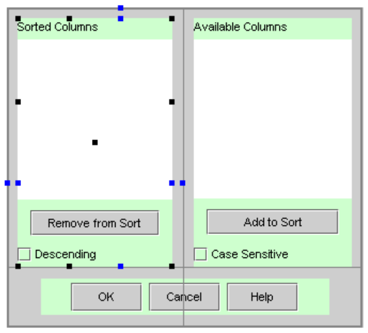
While `GridBagLayout` can accommodate a complex grid, it behaves more successfully and predictably if you organize your components into smaller panels, nested inside the `GridBagLayout` container. These nested panels can use other layouts, including `GridBagLayout`, and can contain additional panels 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 appropriate layouts for specific areas, such as `GridLayout` for 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.

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 5.11 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, as you 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.

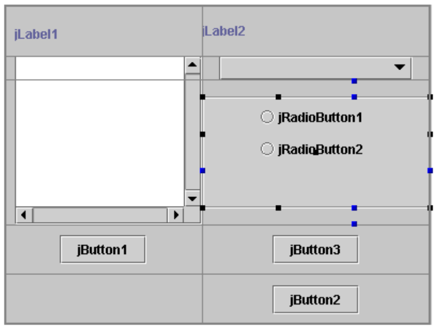
Use the JBuilder visual designer

JBuilder's visual design tools make the time-consuming and risky challenge of using `GridBagLayout` simpler, faster, and more dependable in the following ways.

- 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 5.12 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 "Tips and techniques" on page 94 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 and integrated version control, 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 null or XYLayout

The main advantage to prototyping your UI design in JBuilder's `null` or `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 `null` or `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 `null` or `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 `null` or `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 `null` or `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 the `null` or `XYLayout` 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 `null` or `XYLayout` to `GridBagLayout`, you should leave the inner panels in unconverted 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 `null` or `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 "Tips and techniques" on page 94.
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.

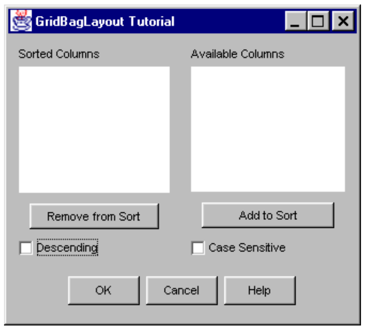
Creating a GridBagLayout in JBuilder

This tutorial uses XYLayout. If you prefer, substitute null layout for XYLayout wherever it's mentioned.

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 5.13 GridBagLayout tutorial UI



The reason for this particular design is precisely because of the complications introduced by adding the odd number of buttons at the bottom. 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 or behave like the images here. 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 states the reasoning behind these modifications.

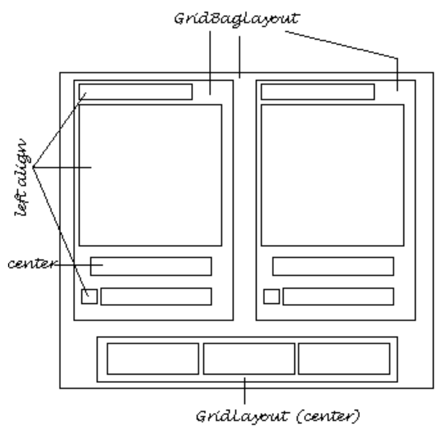
Take your time, and don't be afraid to experiment. The designer makes it easy to try out different things and immediately see the effects. Just save your `Frame1.java` file before you try anything so you can Undo back to that point after you're done.

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 70.

We have already done this step below.

Figure 5.14 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 5.15 Conversion to GridBagLayout by designer

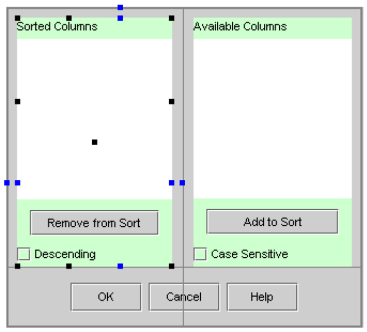


Figure 5.16 Runtime, before resizing

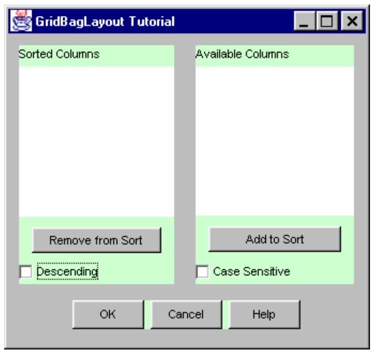
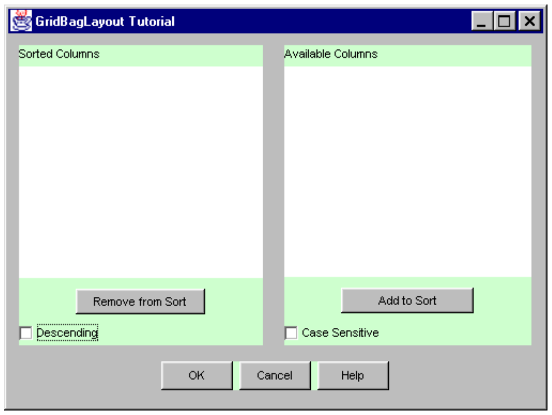


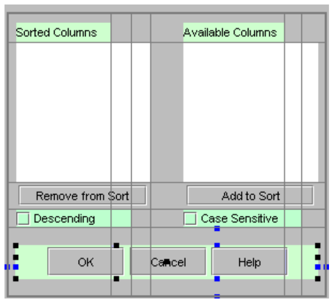
Figure 5.17 Runtime, after resizing



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.

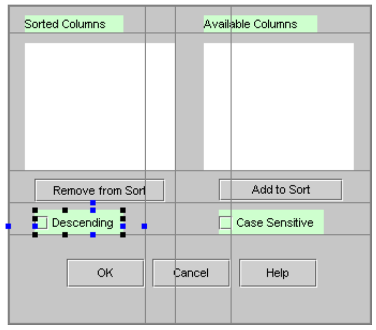
Notice that the conversion created six columns this time. (Again, the background color has been changed on the transparent components so you can see where they end, illustrating how the columns and rows were calculated.)

Figure 5.18 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 5.19 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 5.20 GridBagLayout, no inner panels, before resizing

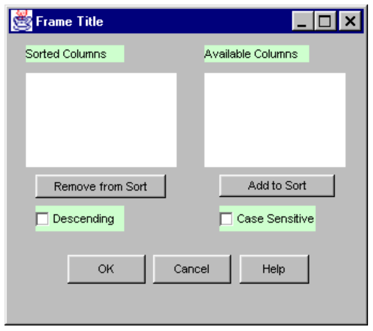
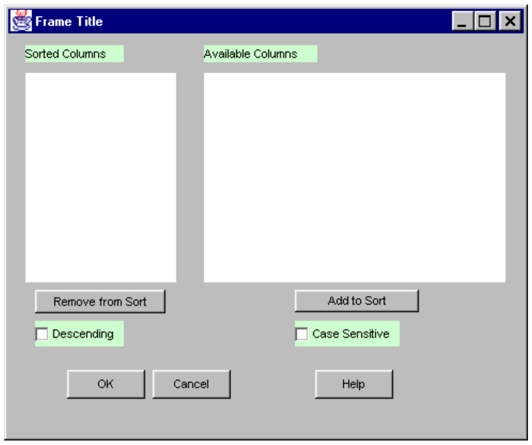


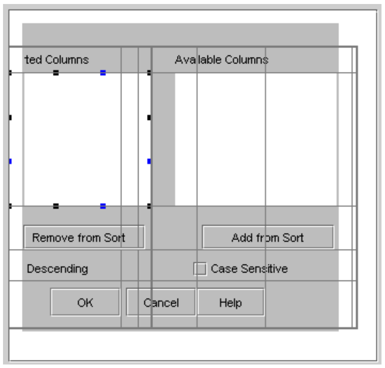
Figure 5.21 GridBagLayout, no inner panels, after resizing



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 5.22 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`:

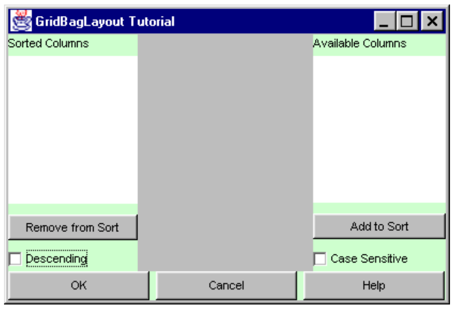
- Group more of the components into nested panels if possible.
- Make your design more symmetrical. Match up the beginning and ending of as many of the components as possible to minimize the number of cells required in the grid.

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 5.23 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|General. Click on the Application icon to open the Application wizard.
4. Select the Application icon and either double-click it or press *Enter*.
5. Accept all defaults making sure no options are checked on step 2, then click Finish.
6. Save the project: choose File|Save Project.

Step 3: Add the components to the containers

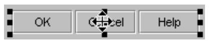
Let's proceed with creating the UI design that uses three nested panels inside a main `GridBagLayout` panel to group the components.

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 5.24 Select middle nib



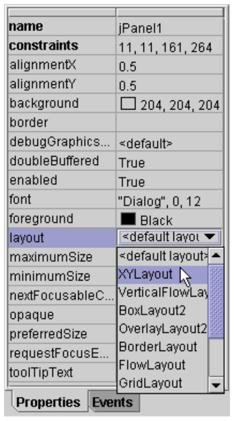
Add the main panel to the UI frame

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 page 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 5.25 Change layout in Inspector

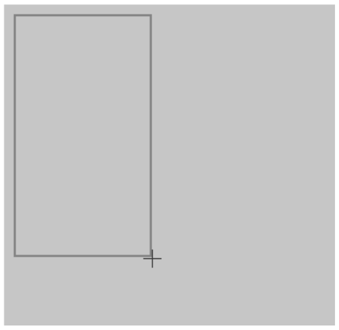


name	jPanel1
constraints	11, 11, 161, 264
alignmentX	0.5
alignmentY	0.5
background	☐ 204, 204, 204
border	
debugGraphics...	<default>
doubleBuffered	True
enabled	True
font	"Dialog", 0, 12
foreground	■ Black
layout	<default layou
maximumSize	<default layout>
minimumSize	XYLayout
nextFocusableC...	VerticalFlowLay
opaque	BoxLayout2
preferredSize	OverlayLayout2
requestFocusE...	BorderLayout
toolTipText	FlowLayout
	GridLayout

Create the left panel and add its components

1 Add a `jPanel` component from the Swing Containers page 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`.

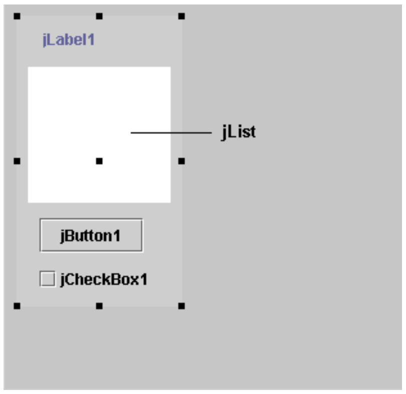
Figure 5.26 GridBagLayout tutorial, jPanel2



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 page, 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.

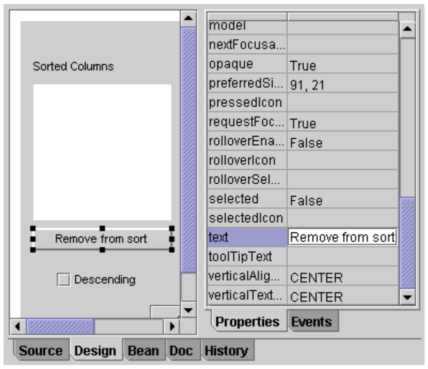
Figure 5.27 GridBagLayout tutorial, jPanel2 components



4 Change the `text` property in the Inspector for these components as follows:

`jLabel1` = "Sorted Columns"
`jButton1` = "Remove from Sort"
`jCheckbox1` = "Descending"

Figure 5.28 Text property



model	
nextFocusa...	
opaque	True
preferredSi...	91, 21
pressedIcon	
requestFoc...	True
rolloverEna...	False
rolloverIcon	
rolloverSel...	
selected	False
selectedIcon	
text	Remove from sort
toolTipText	
verticalAlig...	CENTER
verticalText...	CENTER

5 Match the font for `jLabel1`, `jButton1`, and `jCheckbox1`:

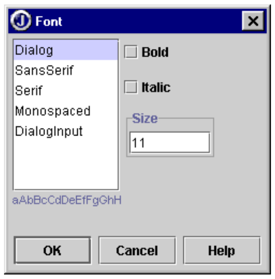
a Hold down the *Ctrl* key, then select `jLabel1`, `jButton1`, and `jCheckbox1` in the component tree.

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 5.29 Font dialog



Size: 11

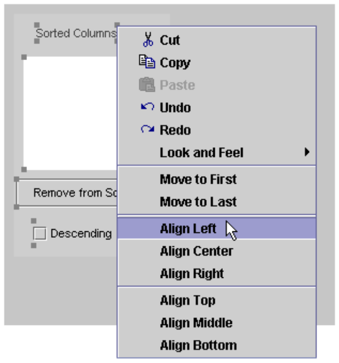
Even up the component sizes and alignment

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 5.30 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 77.

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:

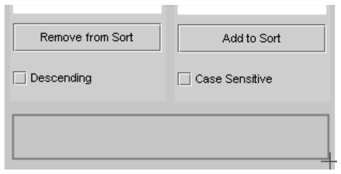
   `jLabel2` = "Available Columns"
   `jButton2` = "Add to Sort"
   `jCheckbox2` = "Case Sensitive"

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 5.31 Draw bottom panel

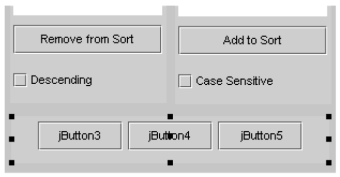


2 Right-click `jPanel4` and choose Align Center.

3 Drop three `jButton` components into `jPanel4`.

4 Change the font for these buttons to match the other components.

Figure 5.32 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 later 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 upper panels to GridBagLayout

Following the good practice of converting inner panels first, then outer ones, we'll start by converting the two upper panels, `jPanel2` and `jPanel3`. The size of these panels will not change when they are converted to `GridBagLayout`(unlike changing to `GridLayout` or `FlowLayout` which resize the container 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). Also make sure the main container ('this') is large enough to give a noticeable margin around its components.

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 5: Convert the lower panel to GridLayout

Now convert the bottom panel:

1 Select `jPanel4` and change its layout to `GridLayout`.

  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 6: Convert the outer panel to GridBagLayout

You're ready to convert jPanel1 to `GridBagLayout`.

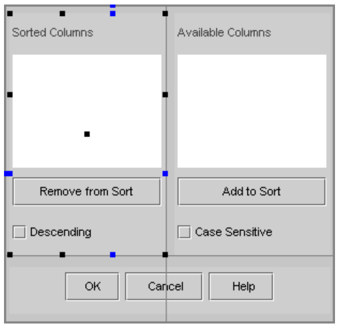
1 First, do a final check to make sure all three panels are aligned well, and that there is a nice amount of space between the inner panels and the edges of `jPanel1`.

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 5.33 Columns after conversion



Step 7: Make final adjustments

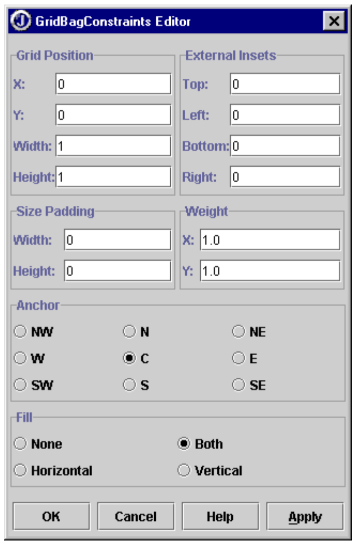
Rather than just give you the constraint values that make this UI design work, we're going to examine each component separately to show why we made the decisions we did. This approach should give you a much better understanding of how the constraints affect the components, cells, and other components.

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 5.34 GridBagConstraints Editor


GridBagConstraints Editor

Grid Position

X:	0
Y:	0
Width:	1
Height:	1

External Insets

Top:	0
Left:	0
Bottom:	0
Right:	0

Size Padding

Width:	0
Height:	0

Weight

X:	1.0
Y:	1.0

Anchor

NW N NE
W [✓ C] E
SW S SE

Fill

None [✓ Both]
Horizontal Vertical

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 5.35 GridLayout with fill on before resizing

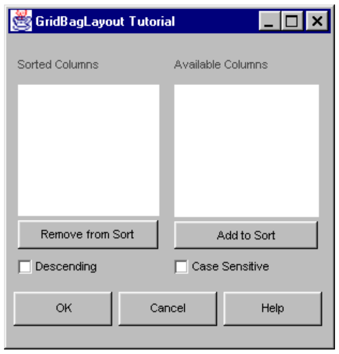
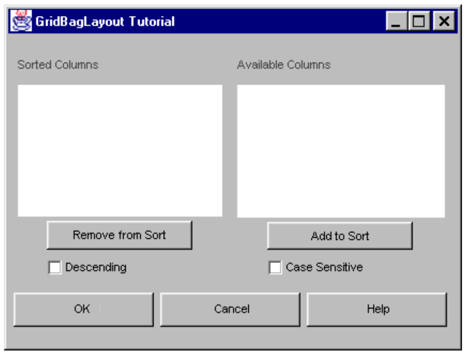


Figure 5.36 GridLayout with fill on after resizing



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`.

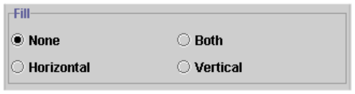
fill

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 5.37 fill constraints


Fill
(● None) ○ Both
○ Horizontal ○ Vertical

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.

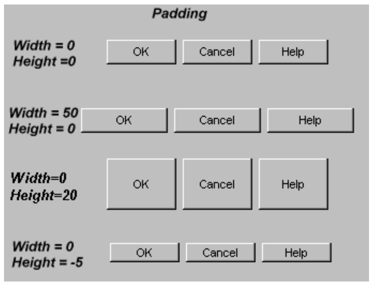
Notice that this action didn't make the buttons shrink to their preferred size. You also need to adjust the padding values to accomplish this.

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.

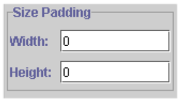
If you like the size of the buttons and the panel at their minimum size, then you don't need to do anything more with the padding. If you want the buttons in the `GridLayout` panel to be larger than their minimum size, you can specify how many additional pixels to add to the panel to accomplish this. You can even make the buttons smaller by using negative values.

Figure 5.38 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 5.39 Padding constraints


Size Padding
Width: 0
Height: 0

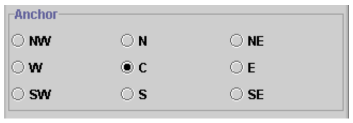
Note Remove padding values for multiple components by multiply selecting the components, right-clicking, and choosing Remove Padding.

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 5.40 anchor constraints



○ NW	○ N	○ NE
○ W	● C	○ E
○ SW	○ S	○ SE

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

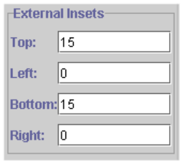
Insets simply define an area between the component and the edges of its display area into which the component cannot enter. It is just like setting margins in a document. No matter how the container is resized, the number of pixels for the `insets` remains constant and work like brakes to keep the component 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 5.41 insets constraints



Top:	15
Left:	0
Bottom:	15
Right:	0

That takes care of the `GridLayout` panel. Now, lets move on to the upper panels.

Upper panels

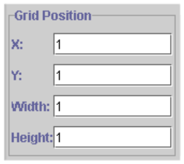
You mostly need to do some clean-up and constraint matching for these panels (`jPanel2` and `jPanel3`) and their components.

gridwidth and gridheight

First, open the GridBagConstraints Editor for each of the components in `jPanel2` and `jPanel3`, 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 5.42 gridwidth and gridheight



X:	1
Y:	1
Width:	1
Height:	1

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 5.43 Add to Sort button without ipadx

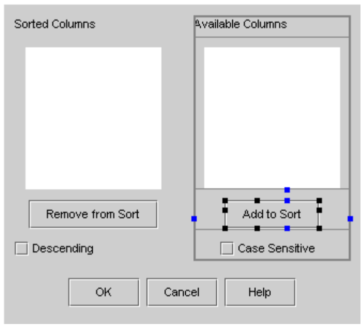
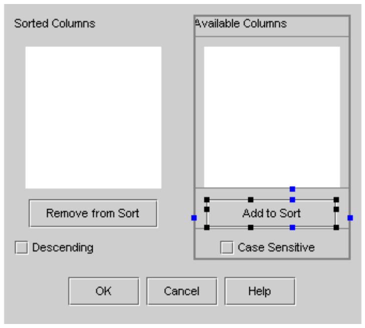


Figure 5.44 Add to Sort button with ipadx



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 5.45 No weight constraints, before resizing

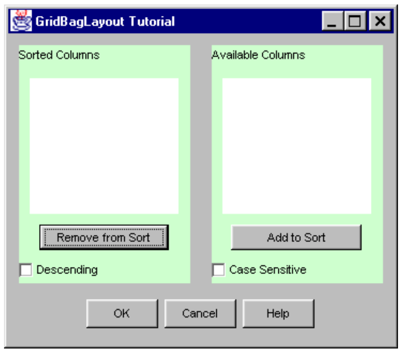
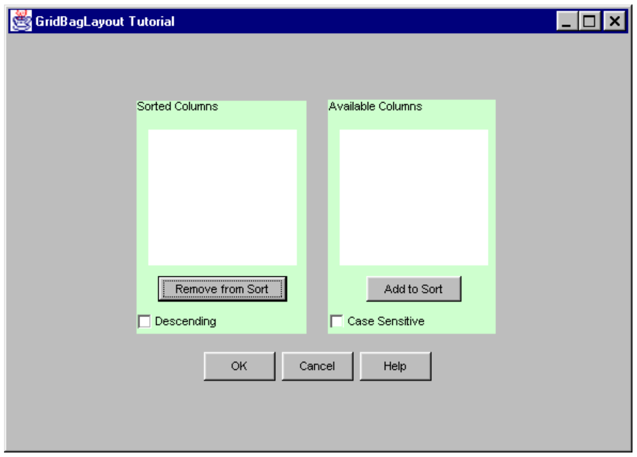


Figure 5.46 No weight constraints, after resizing



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`.

Now save, compile, and run your application. Experiment with resizing it and watch how the components adjust to fit.

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.

Tips and techniques

This tutorial uses XYLayout. If you prefer, substitute null layout for XYLayout wherever it's mentioned.

Setting individual constraints in the designer

anchor

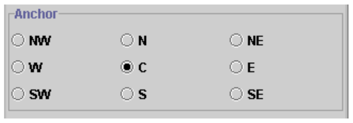
There are two ways to set a component's `anchor` constraint in the designer:

- 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 5.47 Anchor constraints in GridBagConstraints Editor

  Anchor: ( ) NW ( ) N ( ) NE ( ) W [✓ C] ( ) E ( ) SW ( ) S ( ) SE

  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

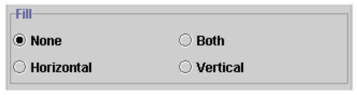
The fastest way to specify the `fill` constraint for a component is to use the component's context menu on the design surface.

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 5.48 Fill constraints in the GridBagConstraints Editor

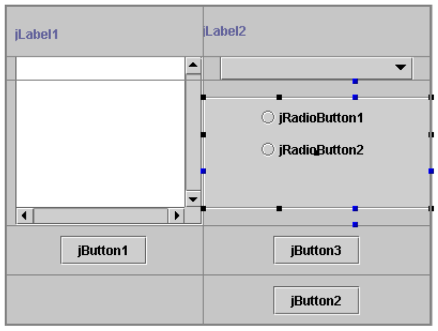

Fill
● None ○ Both
○ Horizontal ○ Vertical

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 around the selected container.

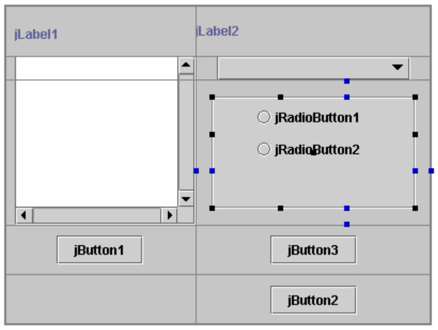
Figure 5.49 Insets set to zero on top and bottom of component



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.

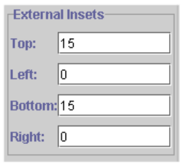
Figure 5.50 Insets greater than zero on right and left sides of component



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 5.51 Insets in the GridBagConstraints Editor



External Insets
Top: 15
Left: 0
Bottom: 15
Right: 0

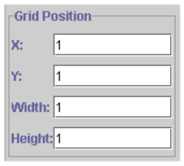
Note While negative Inset values are legal, they can cause components to overlap adjacent components and are not recommended.

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 5.52 Gridwidth and gridheight in the GridBagConstraints Editor



Grid Position
X: 1
Y: 1
Width: 1
Height: 1

- 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`.

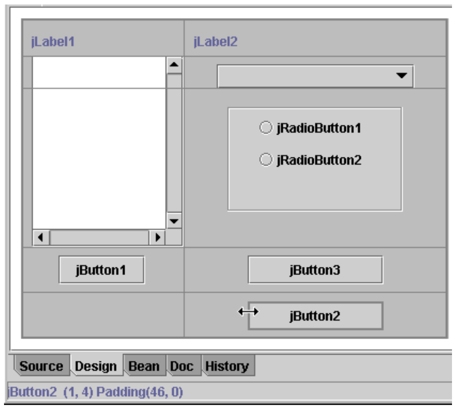
You can also use the mouse to change the `gridwidth` or `gridheight` by sizing the component into adjacent empty cells (dragging a black sizing nib across the cell border.)

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.

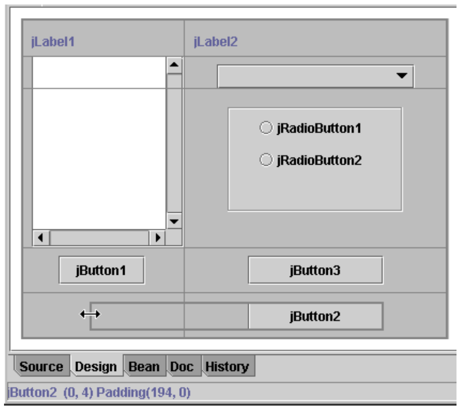
If you drag the sizing nib beyond the edge of the cell into an empty adjacent cell, the component occupies both cells (the `gridwidth` or `gridheight` values increase by one cell).

Figure 5.53 Before: jButton2 gridwidth = 1 cell, ipadx = 46



jButton2 (1, 4) Padding(46, 0)

Figure 5.54 After: jButton2 gridwidth = 2 cells, ipadx increased to 194



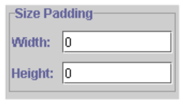
jButton2 (0, 4) Padding(194, 0)

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 5.55 Size Padding in the GridBagConstraints Editor



Size Padding
Width: 0
Height: 0

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

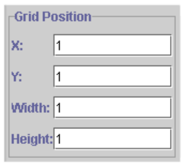
You can use the mouse to specify which cell the upper left corner of the component occupies. Simply click near the upper left corner of the component and drag it into a new cell. When moving components that take up more than one cell, be sure to click in the upper left cell when you grab the component 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 5.56 Grid Position in the GridBagConstraints Editor



X: 1
Y: 1
Width: 1
Height: 1

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.

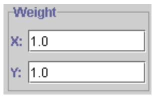
To remove the weight constraints (set them to zero), right-click the component and choose Remove Weights. You can do this for multiple components in a container: hold down the *Shift* key when selecting the components, then right-click and choose Remove Weights.

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 5.57 Weight constraints in the GridBagConstraints Editor



X: 1.0
Y: 1.0

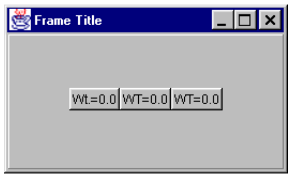
Important `weight` constraints can sometimes make the sizing behavior in the designer difficult to predict. Therefore, setting these constraints should be the last step in designing `GridBagLayout`.

Behavior of weight constraints

Below are some examples of how weight constraints affect the behavior of components:

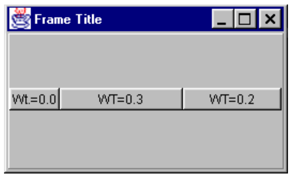
- If all the components have weight constraints of zero in a single direction, the components clump together in the center of the container for that dimension and don't expand beyond their preferred size. `GridBagLayout` puts any extra space between its grid of cells and the edges of the container.

  Figure 5.58 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 5.59 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.

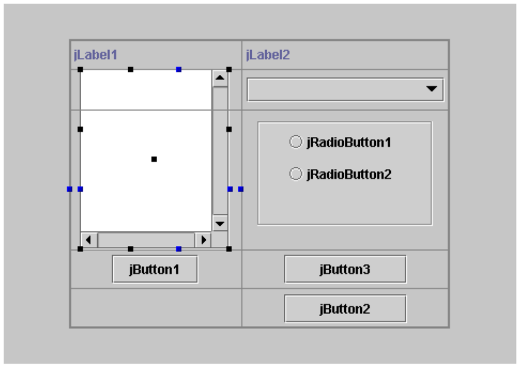
  For example,

  - 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.

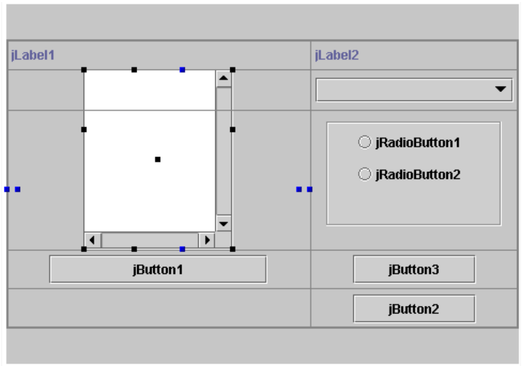
Figure 5.60 GridBagLayout with zero weight constraints on all components



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 5.61 jScrollPane1 with weightx=1.0, no fill constraints

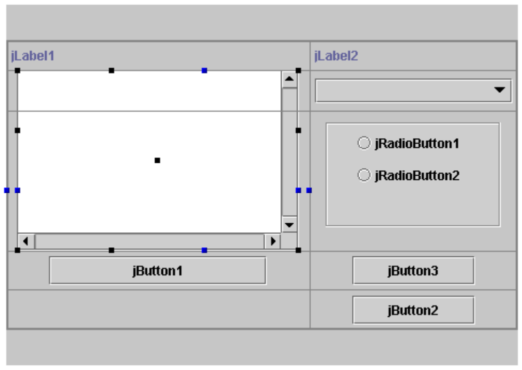


Note If there is more space than you like inside the cells after adding weight to the components, decrease the size of the UI frame until the amount of extra space is what you want. To do this, select the `frame` in the component tree, then click on its black sizing nibs and drag the `frame` to the desired size.

this

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 5.62 jScrollPane1 with weightx=1.0, fill=Horizontal

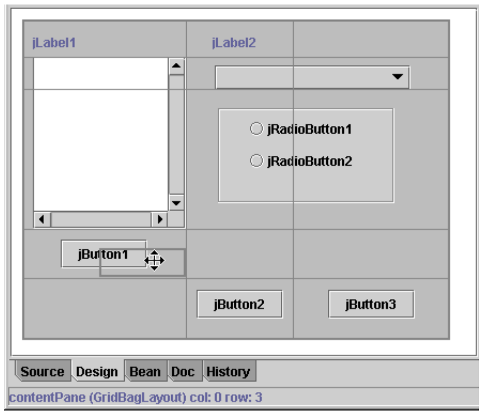


Important If one component in a column has a `weightx` value, `GridBagLayout` gives the whole column that value. Conversely, if one component in a row has a `weighty` value, the whole row is assigned that value.

Using drag and drop to edit constraints

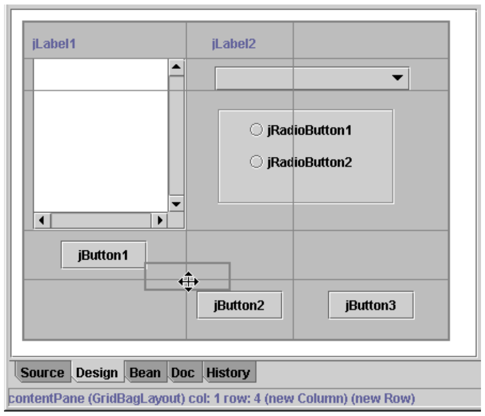
When you drag a component in the designer, the outline of the component indicates where the component will drop. The status bar indicates which column and row the component is in and if the target cell is occupied, what new column and/or row will be created to accommodate the component in its new position.

Figure 5.63 Component being dragged in current cell in the designer



contentPane (GridBagLayout) col: 0 row: 3

Figure 5.64 Component being dragged to new, occupied cell in the designer



contentPane (GridBagLayout) col: 1 row: 4 (new Column) (new Row)

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 5.65 Before moving jButton2, with fill constraint set to HORIZONTAL

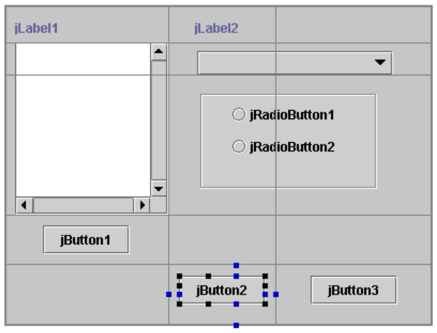
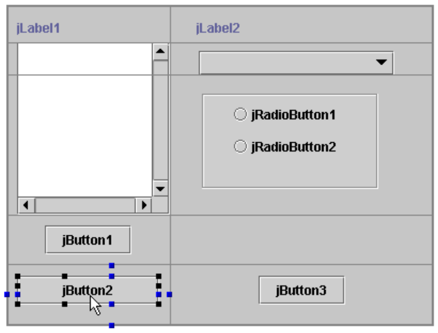


Figure 5.66 Moving jButton2 to same size or larger cell



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 5.67 Before moving jButton3, with fill constraint set to HORIZONTAL

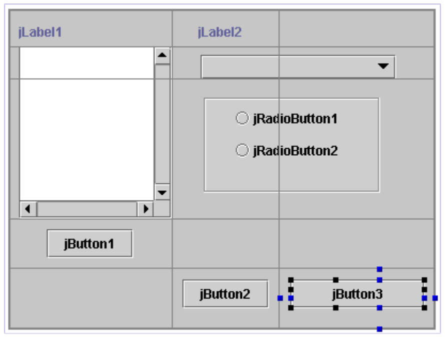
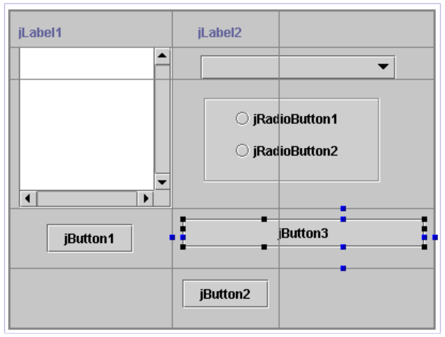


Figure 5.68 Moving jButton3 to smaller cell above jButton2



See "Dragging a component to an occupied cell" below.

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.

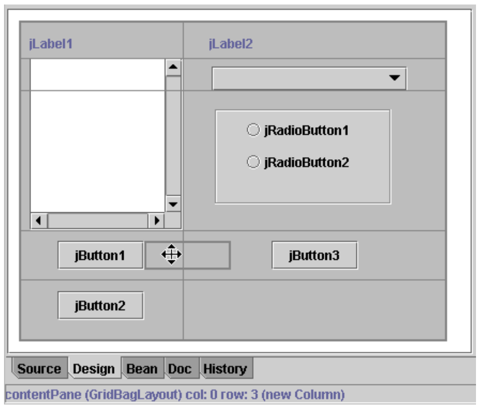
It's especially important to understand this action and resulting behavior. If you are using the designer to build up a new UI in `GridBagLayout`, rather than in `XYLayout` or `null` layout, much of the time the grid will be full when you drop a new component onto it, resulting in the creation of new 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`:

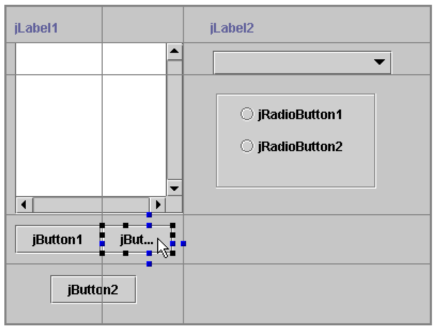
- It inserts a new column between the existing two columns.
- It increases the `gridwidth` for `jLabel1`, the `jScrollPane1`, and `jButton2` to two columns instead of one.
- It removes the right `inset` from `jButton1` and the left `inset` from `jButton3`.

Figure 5.69 Dragging jButton3 into cell with jButton1



contentPane (GridBagLayout) col: 0 row: 3 (new Column)

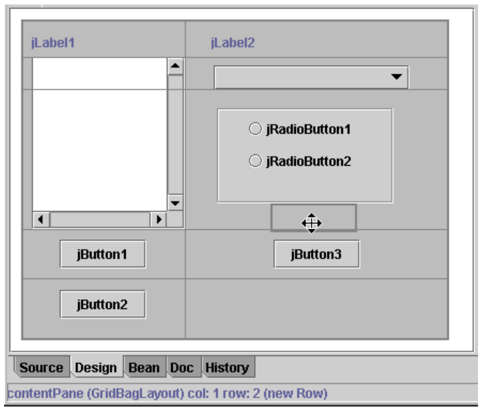
Figure 5.70 After moving the button horizontally



If `jButton3` is dragged into the cell with the radio button panel (`jPanel1`):

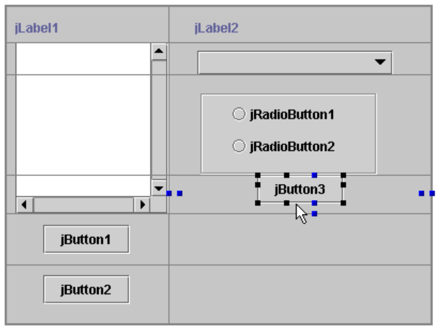
- It inserts a new row before `jButton1` by splitting row 2 in half (occupied by the `jPanel1` and the bottom part of the `jScrollPane1`).
- It increases the `gridheight` of the `jScrollPane1` from two to three rows.
- It takes away space from the large row occupied by the `jScrollPane1` and the `jPanel1`.
- It removes the bottom `inset` for `jPanel1` and the top inset for `jButton3`.

Figure 5.71 Dragging jButton3 into cell with jPanel1



contentPane (GridBagLayout) col: 1 row: 2 (new Row)

Figure 5.72 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 5.73 Dragging jPanel1 into a small, empty cell

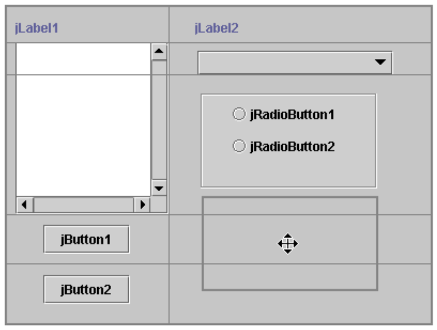
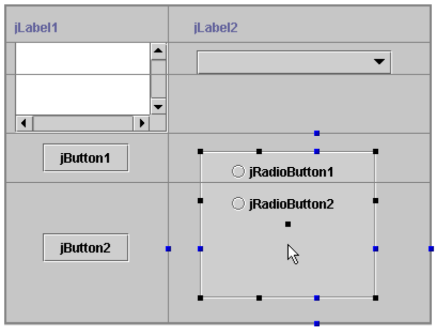


Figure 5.74 After dragging jPanel1



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 5.75 Before dragging top sizing nib of jButton2 into empty cell above

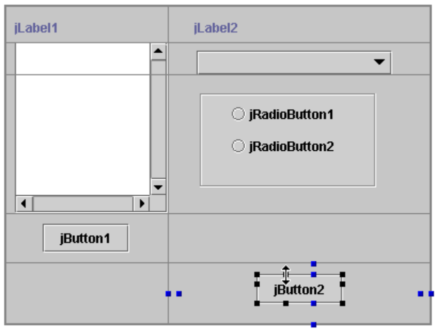


Figure 5.76 During drag

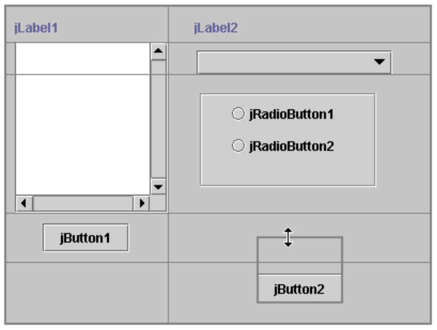
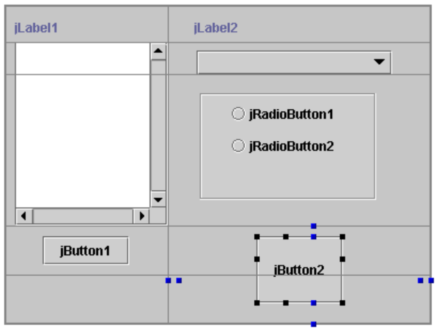


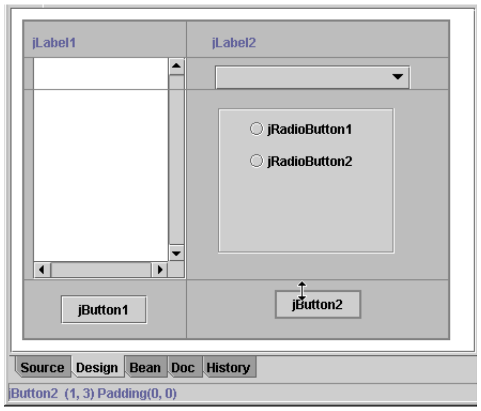
Figure 5.77 After drag, jButton2 has gridheight of 2 with insets unchanged



Dragging the black sizing nibs into an adjacent occupied cell

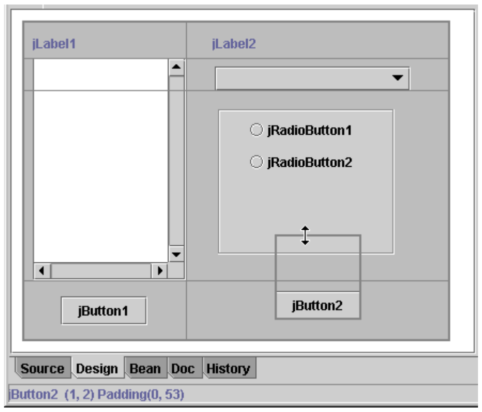
Dragging a component's black sizing nib into an adjacent occupied cell increases the component's `ipadx` and `ipady` constraint values. Notice how the status bar indicates this in the example below.

Figure 5.78 Before dragging, jButton2 has zero padding



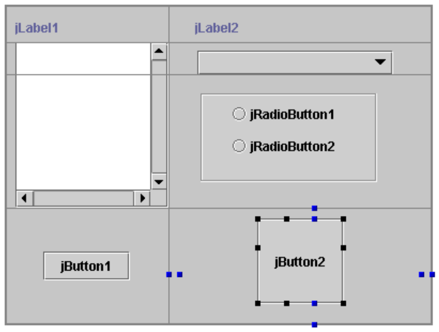
jButton2 (1, 3) Padding(0, 0)

Figure 5.79 Dragging sizing nib into occupied cell increases padding



jButton2 (1, 2) Padding(0, 53)

Figure 5.80 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 5.81 Clicking above jButton1 creates a new row for jButton3 above jButton1

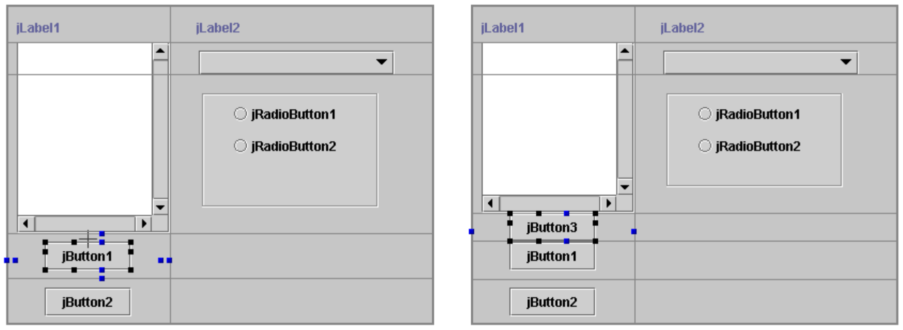


Figure 5.82 Clicking to the left of jButton1 creates a new column for jButton3 to the left of jButton1

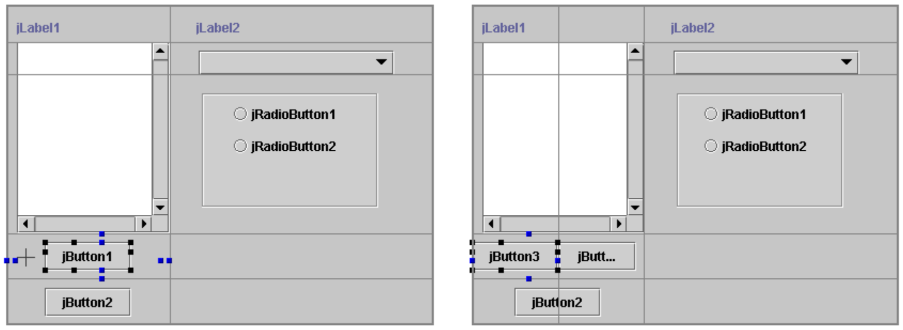


Figure 5.83 Clicking below jButton1 creates a new row for jButton3 below jButton1

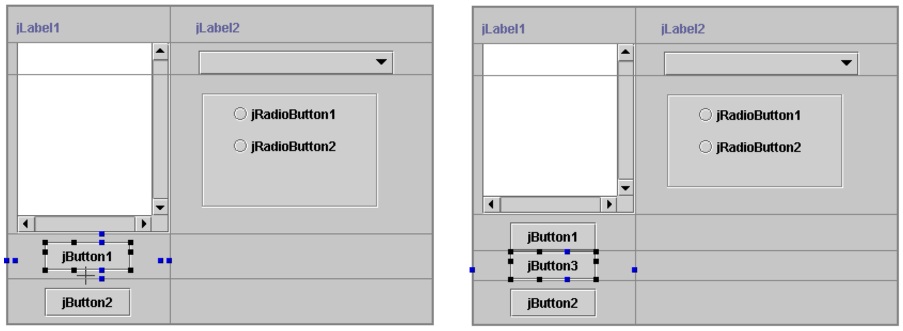
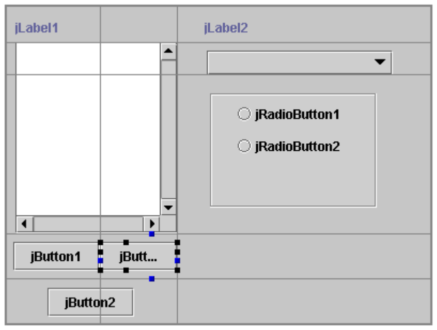


Figure 5.84 Clicking to the right of jButton1 creates a new column for jButton3 to the right of jButton1



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 Part 2.

```
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

http://java.sun.com/j2se/1.3/docs/api/java/awt/GridBagConstraints.html
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.HORIZONAL`	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 5.85 Weight constraints on both panels and lists

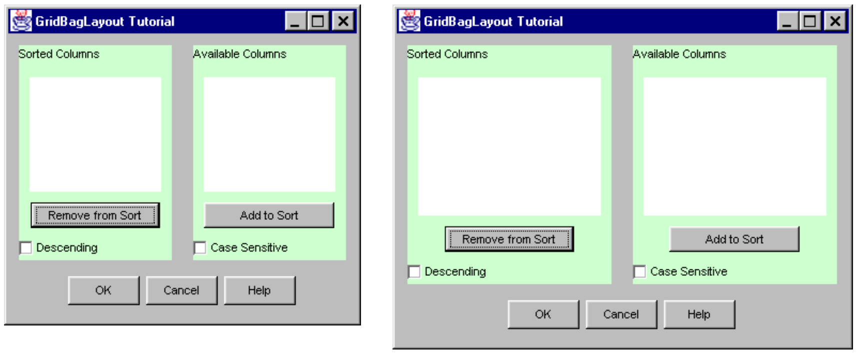


Figure 5.86 Weight constraints on the panels, but not on the lists

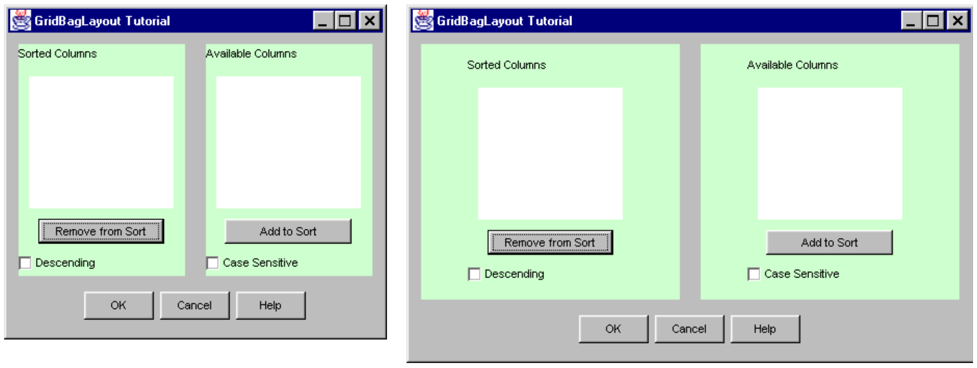
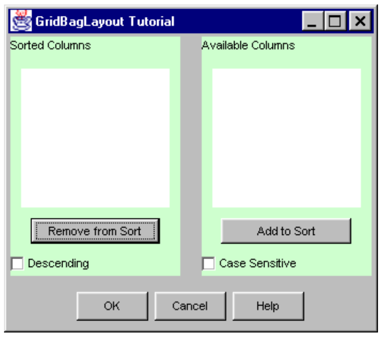


Figure 5.87 Weight constraints on the lists, but not on the panels



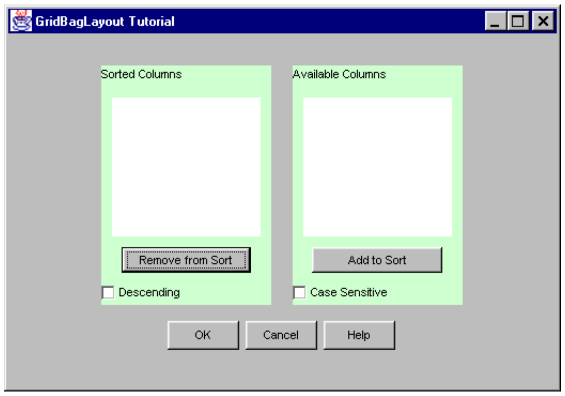


Figure 5.88 Horizontal weight constraints (weightx) only on all four components

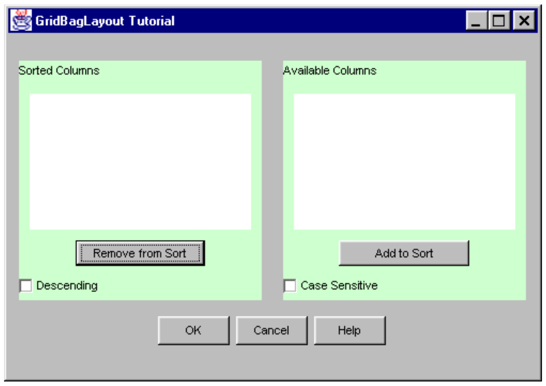


Figure 5.89 Vertical weight constraints (weighty) only on all four components

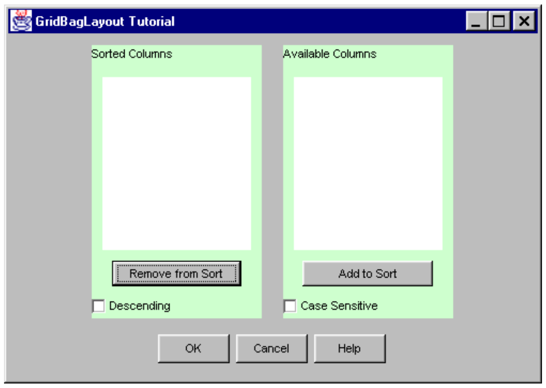
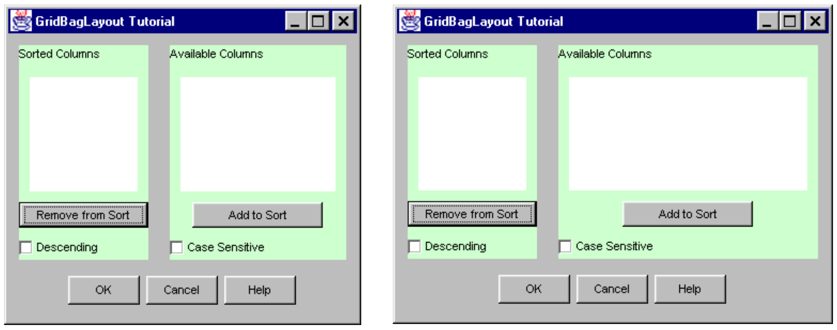


Figure 5.90 Weight constraints on only one panel and list component in the row



Index