CONTENTS

Chapter 12. Layout Managers

Topics in This Chapter

- How layout managers simplify interface design
- The FlowLayout manager
- The BorderLayout manager
- The GridLayout manager
- The CardLayout manager
- The GridBagLayout manager
- The BoxLayout manager
- Hand-positioning components
- Strategies for using layout managers effectively

When a `Container` is created in the Java programming language, the container automatically gets an associated helper object known as a *layout manager* to give sizes and positions to the components inside it. The layout manager is intended to free the programmer from the burden of positioning each component pixel-by-pixel when the components may be different sizes on different platforms, when the main windows may be interactively resized or customized based on parameters in the HTML file, or when the design changes several times during development.

Although this idea is a good one, in practice the built-in layout managers are good for basic layouts but not flexible enough for many complex arrangements. However, by using nested containers, each of which has its own layout manager, you can achieve relatively complex layouts even with the simplest layout managers. Besides, you can always turn the layout manager off if you'd rather do things by hand. You can also design your own layout manager to fit your specific requirements or favored look.

To use a layout manager, you first associate a manager with the container by using `setLayout` (or by simply accepting the window's default), then insert components into the window with `add`. In an applet, setting the layout manager is usually performed in `init`. In an application, the layout manager is usually set in the constructor.

In this chapter, we first cover the five AWT layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, and `GridBagLayout` and then cover the new Swing layout

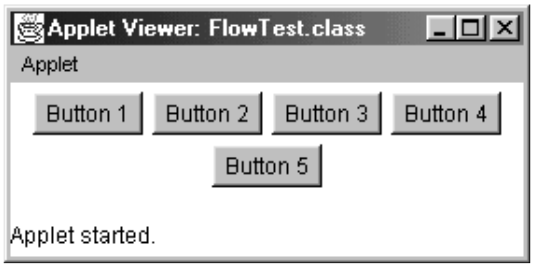
manager, `BoxLayout`. We then discuss turning off the layout manager and tips for using layout managers effectively.

12.1 The FlowLayout Manager

The `FlowLayout` manager resizes components to their preferred (natural) size, then arranges them in rows in the window. The first component is placed in the leftmost position in the top row, the second component next, and so on until the next component would not fit in the current row, in which case the process is repeated in the next row for the remaining components. By default, each row is centered and five pixels are left between components in a row and between rows. The constructor options described later in this section let you make left-aligned or right-aligned rows and let you change the spacing between components. `FlowLayout` is the default layout manager for a `Panel`, `JPanel`, and `Applet`.

Listing 12.1 shows an example that places five buttons in a window that is wide enough to hold only four. As Figure 12-1 shows, the group containing the first four is centered on the top row of the applet, and the last button is centered in a row directly underneath. Note that because `FlowLayout` is the default layout manager for `Applet`, there is no need for an explicit `setLayout` call. For containers using other layout managers, you would use `setLayout (new FlowLayout())` to stipulate that `FlowLayout` be used.

Figure 12-1. `FlowLayout` puts components in rows, moving to a new row only when the component won't fit in the current row.



Listing 12.1 `FlowTest.java`

```
import java.applet.Applet;
import java.awt.*;

/** FlowLayout puts components in rows. */

public class FlowTest extends Applet {
  public void init() {
    for(int i=1; i<6; i++) {
      add(new Button("Button " + i));
    }
  }
}
```

FlowLayout Constructor Options

public FlowLayout()

This constructor builds a `FlowLayout` layout manager that centers each row and

keeps five pixels between entries in a row and between rows.

public FlowLayout(int alignment)

This constructor builds a `FlowLayout` with left-aligned, right-aligned, or centered rows, depending on the value of the alignment argument (`FlowLayout.LEFT`, `FlowLayout.RIGHT`, or `FlowLayout.CENTER`). The horizontal and vertical gap is five pixels.

public FlowLayout(int alignment, int hGap, int vGap)

This constructor builds a `FlowLayout` that uses the specified alignment, keeps `hGap` pixels between entries in a row, and reserves `vGap` pixels between rows.

Other FlowLayout Methods

The following methods are available:

public int getAlignment()

public void setAlignment(int alignment)

These methods let you look up or modify the row alignment. As in the constructor, the alignment can be `FlowLayout.LEFT`, `FlowLayout.RIGHT`, or `FlowLayout.CENTER`.

public int getHgap()

public void setHgap(int hGap)

These methods let you look up or change the empty space kept between entries in a row. The default is 5.

public int getVgap()

public void setVgap(int vGap)

These methods let you look up or change the amount of empty space between rows. The default is 5.

The default `FlowLayout` instance used by panels is stored in a `static` (shared) variable in the `Panel` class. However, you have to exercise caution to avoid unintended side effects. Although the shared `FlowLayout` is stored in a `final` variable, that only means that the *reference* is constant; the reference cannot be redirected to refer to a new object. A final reference to an object does not mean that the *fields* of the object are unmodifiable. In this regard, the Java programming language has the equivalent of C++'s `const` pointers, but not the equivalent of `const` objects. For instance, if you want to change the horizontal gap in a `Panel`, you should do so with

```
Panel p = new Panel();
p.setLayout(new FlowLayout(FlowLayout.CENTER,10,5));
```

not

```
Panel p = new Panel();
```

```
FlowLayout layout = (FlowLayout)p.getLayout();
layout.setHgap(10);
```

You might think that the latter is more efficient because a new `FlowLayout` instance is not allocated, but the consequences of this fact mean that changes to the shared `FlowLayout` instance can affect other panels. Any other panel using the default layout manager will now have a horizontal gap of 10.

12.2 The BorderLayout Manager

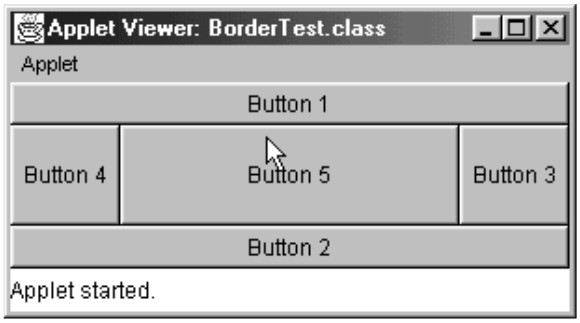
`BorderLayout` divides the window into five sections: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`. You add a component to a region by using a version of the `add` method that takes two arguments instead of the normal one. The first argument is the component, and the second argument is a string naming the region that should hold the component, as follows:

```
add(buttonForTop, BorderLayout.NORTH);
add(scrollbarForRightSide, BorderLayout.EAST);
add(panelForRemainingSpace, BorderLayout.CENTER);
```

The role of the layout manager is to provide sizes and positions for the components. Components added to `NORTH` or `SOUTH` are resized to their preferred height, with a width equal to the `Container`, and then placed at the top or bottom of the window. Components added to `EAST` or `WEST` are resized to take the full height of the `Container` (minus any space taken by `NORTH` and `SOUTH`) and to be their preferred widths. A component in the `CENTER` region is expanded to take whatever space is remaining. The `BorderLayout` constructor also allows specification of the gaps between the areas; the default is 0. `BorderLayout` is the default layout manager for `Frame`, `Dialog`, and `Window` and for the content pane of `JApplet`, `JFrame`, `JDialog`, and `JWindow`.

Listing 12.2 shows an applet that uses `BorderLayout` and that places a button in each of the five regions. As Figure 12-2 shows, these buttons are stretched according to the size of the window, rather than remaining their preferred size as when `FlowLayout` is used.

Figure 12-2. `BorderLayout` divides the window into five regions.



Listing 12.2 `BorderTest.java`

```
import java.applet.Applet;
import java.awt.*;

/** An example of BorderLayout. */
```

```
public class BorderTest extends Applet {
  public void init() {
    setLayout(new BorderLayout());
    add(new Button("Button 1"), BorderLayout.NORTH);
    add(new Button("Button 2"), BorderLayout.SOUTH);
    add(new Button("Button 3"), BorderLayout.EAST);
    add(new Button("Button 4"), BorderLayout.WEST);
    add(new Button("Button 5"), BorderLayout.CENTER);
  }
}
```

The single most likely error you will make when first using `BorderLayout` is forgetting to use:

```
add(component,BorderLayout.REGION);
```

and instead doing the more familiar but incorrect

```
add(component);
```

If you don't specify a region, the component is added to the `CENTER` region. Since entries in the `CENTER` occupy all available space, only the topmost component (last one added) is displayed. No warning is given when compiling or running.

Core Warning

If you forget to specify a region when adding a component to a window that uses `BorderLayout`, the component is added to the `CENTER` region.

Also remember that you should have at most one component per region when using `BorderLayout`. Otherwise, the top component will obscure the one underneath. If you want several components in a region, group them in a `Panel` and then put the `Panel` in the desired region.

Core Approach

Add at most one component to each region of a `BorderLayout`. To have multiple components in a region, group them in a `Panel` or other `Container`.

BorderLayout Constructor Options

public BorderLayout()

This constructor builds a `BorderLayout` object with regions that touch each other.

public BorderLayout(int hGap, int vGap)

This constructor builds a `BorderLayout` object that reserves `hGap` empty pixels between the `WEST` and `CENTER` regions and between `CENTER` and `EAST`. The constructor also keeps `vGap` blank pixels between the `NORTH` and `CENTER` regions and between `CENTER` and `SOUTH`.

Other BorderLayout Methods

You should be careful not to use `setHgap` or `setVgap` on the return value of `getLayout` when applied to a `Frame` or `Dialog` that has no explicit layout manager set, since the change will affect other containers that use `BorderLayout` by default. The same recommendation is also true for the `contentPane` of a `JFrame`, `JApplet`, `JWindow`, and `JDialog`.

public int getHgap()

public void setHgap(int hGap)

These methods look up and specify the empty space between horizontally adjacent regions.

public int getVgap()

public void setVgap(int vGap)

These methods look up and specify the empty space between vertically adjacent regions.

public float getLayoutAlignmentX(Container c)

public float getLayoutAlignmentY(Container c)

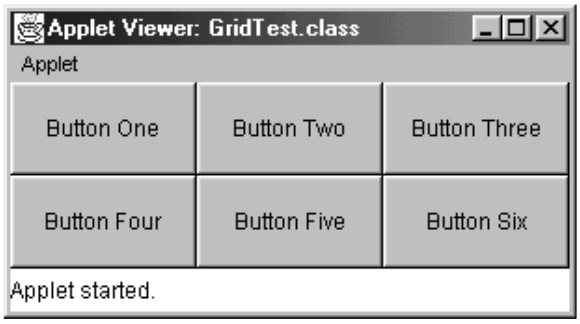
These methods tell you how the Container should be aligned. For more details, see getAlignmentX and getAlignmentY in Section 13.2 (The Component Class).

12.3 The GridLayout Manager

The `GridLayout` manager divides the window into equal-sized rectangles based upon the number of rows and columns specified. Items are placed into the cells left to right, top to bottom, based upon the order in which they are added. Each component is resized to fit into its grid cell without regard to its current or preferred size. A constructor option lets you specify the gaps between the rows and columns.

Listing 12.3 presents an applet divided into two rows and three columns, with a button placed into each cell. Figure 12-3 shows the result.

Figure 12-3. `GridLayout` divides the window into equal-sized rectangles.



Listing 12.3 `GridTest.java`

```
import java.applet.Applet;
```

```
import java.awt.*;

/** An example of GridLayout. */

public class GridTest extends Applet {
  public void init() {
    setLayout(new GridLayout(2,3)); // 2 rows, 3 cols
    add(new Button("Button One"));
    add(new Button("Button Two"));
    add(new Button("Button Three"));
    add(new Button("Button Four"));
    add(new Button("Button Five"));
    add(new Button("Button Six"));
  }
}
```

GridLayout Constructor Options

public GridLayout()

This constructor creates a single row with one column allocated per component.

public GridLayout(int rows, int cols)

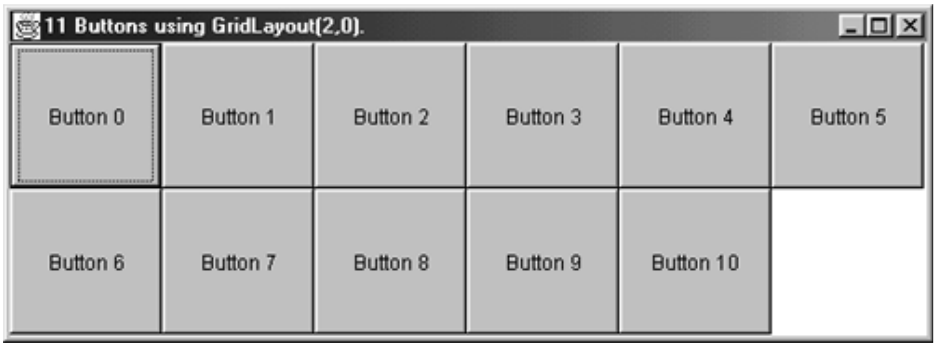
This constructor creates a `GridLayout` object that divides the window into the specified number of rows and columns, with each grid cell flush against the neighboring cell. Either `rows` or `cols` (but not both) can be 0. If you specify 0 for the number of rows, the Java platform will try to set the number of rows so that each column has approximately the same number of elements. Similarly, if you specify 0 for the column count, the Java platform will try to choose the number of columns so that each row gets approximately an even number of components. To illustrate this, Listing 12.4 places 11 buttons in a window that specifies zero columns, taking the number of rows as a command-line argument. See Listing 14.1 for `WindowUtilities.java`.

Listing 12.4 `ElevenButtons.java`

```
import java.awt.*;
import javax.swing.*;

/** This illustrates the effect of specifying 0 for the number
 *  of columns. The number of rows is read from the command line
 *  (default 2), and the column number is chosen by the system
 *  to get as even a layout as possible.
 */

public class ElevenButtons extends JPanel {
  public ElevenButtons(int numRows) {
    setLayout(new GridLayout(numRows, 0));
    for(int i=0; i<11; i++) {
      add(new JButton("Button " + i));
    }
  }
```

```
  public static void main(String[] args) {
    int numRows = 2;
    if (args.length > 0) {
      numRows = Integer.parseInt(args[0]);
    }
    String title = "11 Buttons using GridLayout(" +
                   numRows + ",0).";
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(new ElevenButtons(numRows),
                                 550, 200, title);
  }
}
```

Figure 12-4. When specifying two rows and zero columns, the Java platform chooses 6 columns for an 11-component window.



public GridLayout(int rows, int cols, int hGap, int vGap)

This constructor divides the window into the specified number of rows and columns, leaving `hGap` and `vGap` empty pixels between columns and rows. Either the row count or the column count can be 0, but not both. A value of 0 is interpreted as "any number," as illustrated with the description of the previous constructor.

Other GridLayout Methods

public int getRows()

public void setRows(int rows)

These methods let you look up and modify the number of rows.

public int getColumns()

public void setColumns(int cols)

These methods let you read and change the number of columns.

public int getHgap()

public void setHgap(int hGap)

These methods report and specify the empty space between columns.

public int getVgap()

public void setVgap(int vGap)

These methods report and specify the empty space between rows.

12.4 The CardLayout Manager

The `CardLayout` manager stacks components on top of each other, displaying one at a time. To use this layout manager, you create a `CardLayout` instance, telling the window to use the manager but also saving the reference in a variable:

```
Panel cardPanel;
CardLayout layout;
...
layout = new CardLayout();
cardPanel.setLayout(layout);
```

Don't forget to save a reference to the layout manager; you'll need the reference later. You then associate a name with each component you add to the window, as below:

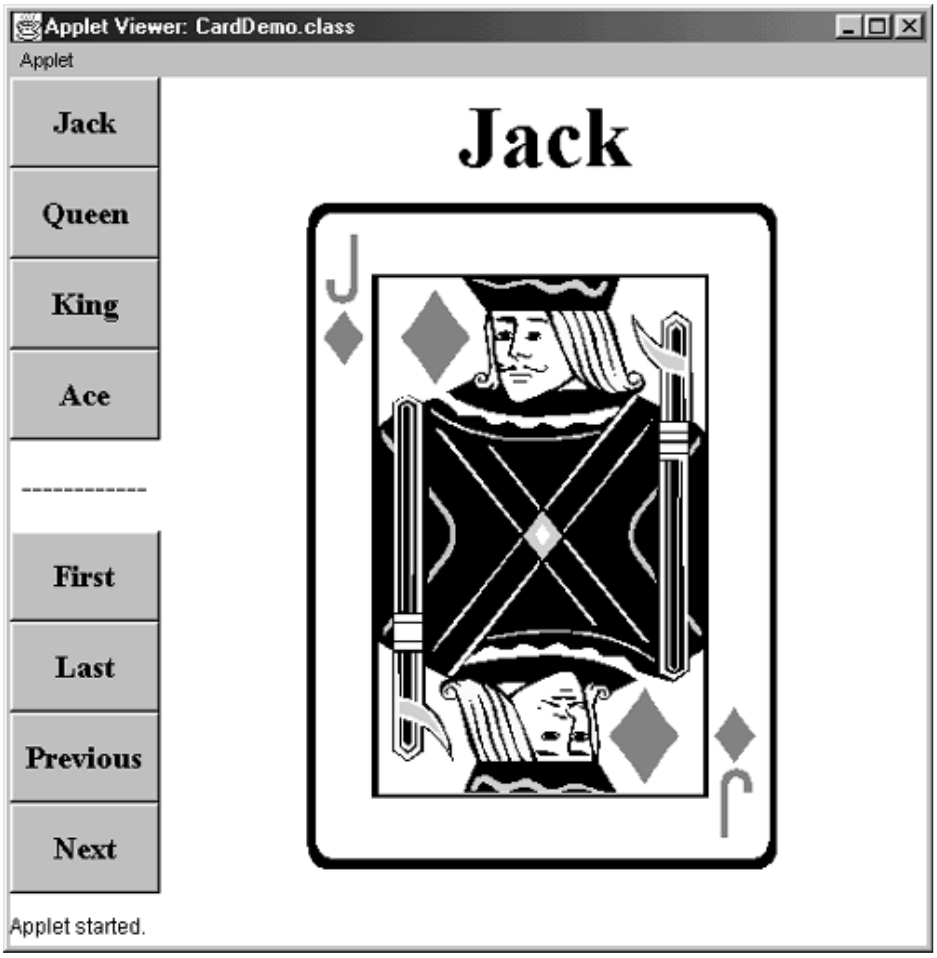
```
cardPanel.add(component1, "Card 1");
cardPanel.add(component2, "Card 2");
...
```

The first component added is shown by default. You tell the layout manager to display a particular component in a number of ways: for example, by specifying the name with a call to `show` or by calling `first`, `last`, `previous`, or `next`. In all cases, the `Container` using the layout manager must be supplied as an argument, as follows:

```
layout.show(cardPanel, "Card 1");
layout.first(cardPanel);
layout.next(cardPanel);
```

By way of example, consider Listing 12.5. It places a `Panel`, using `CardLayout`, in the right-hand region of the window and a column of buttons in the left-hand region. The `Panel` contains four versions of a `CardPanel` (Listing 12.6), which is a `Panel` that holds a `Label` and an `ImageLabel` (you can find the source code for `ImageLabel.java` at http://www.corewebprogramming.com/) showing a picture of a playing card. Depending on which button is selected, a different card is shown. Figure 12-5 shows one of the four possible configurations.

Figure 12-5. `CardLayout` stacks components above each other.



Listing 12.5 `CardDemo.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/** An example of CardLayout. The right side of the window holds
 *  a Panel that uses CardLayout to control four possible
 *  subpanels (each of which is a CardPanel that shows a
 *  picture of a playing card). The buttons on the left side of
 *  the window manipulate the "cards" in this layout by
 *  calling methods in the right-hand panel's layout manager.
 */

public class CardDemo extends Applet implements ActionListener {
  private Button first, last, previous, next;
  private String[] cardLabels = { "Jack","Queen","King","Ace" };
  private CardPanel[] cardPanels = new CardPanel[4];
  private CardLayout layout;
  private Panel cardDisplayPanel;

  public void init() {
    setBackground(Color.white);
    setLayout(new BorderLayout());
```

```
    addButtonPanel();
    addCardDisplayPanel();
  }

  private void addButtonPanel() {
    Panel buttonPanel = new Panel();
    buttonPanel.setLayout(new GridLayout(9, 1));
    Font buttonFont = new Font("SansSerif", Font.BOLD, 18);
    buttonPanel.setFont(buttonFont);
    for(int i=0; i<cardLabels.length; i++) {
      Button button = new Button(cardLabels[i]);
      button.addActionListener(this);
      buttonPanel.add(button);
    }
    first = new Button("First");
    first.addActionListener(this);
    last = new Button("Last");
    last.addActionListener(this);
    previous = new Button("Previous");
    previous.addActionListener(this);
    next = new Button("Next");
    next.addActionListener(this);
    buttonPanel.add(new Label("------------", Label.CENTER));
    buttonPanel.add(first);
    buttonPanel.add(last);
    buttonPanel.add(previous);
    buttonPanel.add(next);
    add(buttonPanel, BorderLayout.WEST);
  }
  private void addCardDisplayPanel() {
    cardDisplayPanel = new Panel();
    layout = new CardLayout();
    cardDisplayPanel.setLayout(layout);
    String cardName;
    for(int i=0; i<cardLabels.length; i++) {
      cardName = cardLabels[i];
      cardPanels[i] =
        new CardPanel(cardName, getCodeBase(),
                      "images/" + cardName + ".gif");
      cardDisplayPanel.add(cardPanels[i], cardName);
    }
    add(cardDisplayPanel, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent event) {
    Button source = (Button)event.getSource();
    if (source == first)
      layout.first(cardDisplayPanel);
    else if (source == last)
      layout.last(cardDisplayPanel);
    else if (source == previous)
```

```
      layout.previous(cardDisplayPanel);
    else if (source == next)
      layout.next(cardDisplayPanel);
    else
      layout.show(cardDisplayPanel, source.getLabel());
    return;
  }
}
```

Listing 12.6 `CardPanel.java`

```
import java.awt.*;
import java.net.*;

/** A Panel that displays a playing card. This window does
 *  <B>not</B> use CardLayout. Rather, instances of CardPanel
 *  are contained in another window used in the CardDemo
 *  example. It is this enclosing window that uses CardLayout
 *  to manipulate which CardPanel it shows.
 */

public class CardPanel extends Panel {
  private Label name;
  private ImageLabel picture;

  public CardPanel(String cardName,
                   URL directory, String imageFile) {
    setLayout(new BorderLayout());
    name = new Label(cardName, Label.CENTER);
    name.setFont(new Font("SanSerif", Font.BOLD, 50));
    add(name, BorderLayout.NORTH);
    picture = new ImageLabel(directory, imageFile);
    Panel picturePanel = new Panel();
    picturePanel.add(picture);
    add(picturePanel, BorderLayout.CENTER);
    setSize(getPreferredSize());
  }

  public Label getLabel() {
    return(name);
  }

  public ImageLabel getImageLabel() {
    return(picture);
  }
}
```

CardLayout Constructor Options

public CardLayout()

This constructor creates a `CardLayout` instance that places components in the top-

left corner of the window.

public CardLayout(int sideMargins, int topMargins)

This constructor creates a `CardLayout` object that reserves the specified number of empty pixels on the sides and top/bottom of the window.

Other CardLayout Methods

The following methods are available in the `CardLayout` class. Note that the `Container` argument refers to the window that is using the `CardLayout`, not to the card (`Component`) itself.

public void show(Container c, String cardName)

This method displays the card that was given the specified name when added to the `Container`.

public void first(Container c)

public void last(Container c)

These methods display the card that was added first and last, respectively.

public void previous(Container c)

public void next(Container c)

These methods display the previous or next card in the sequence. Unless you use a version of `add` that lets you specify a position in the component array, the sequence is arranged in the order in which components were added.

public int getHgap()

public void setHgap(int hGap)

These methods report and specify the empty space reserved at the left and right sides. The names `getSideMargins` and `setSideMargins` might have been clearer in this case, but these names were chosen for consistency with other layout managers.

public int getVgap()

public void setVgap(int vGap)

These methods report and specify the empty space reserved at the top and bottom of the window. The names `getTopMargins` and `setTopMargins` might have been clearer in this case, but these names were chosen for consistency with other layout managers.

public float getLayoutAlignmentX(Container c)

public float getLayoutAlignmentY(Container c)

These methods tell you how the `Container` should be aligned. For more details, see `getAlignmentX` and `getAlignmentY` in Section 13.2 (The Component Class).

12.5 GridBagLayout

The `GridBagLayout` manager is about three times more flexible than any of the other standard layout managers. Unfortunately, `GridBagLayout` is also about *nine* times harder to use. The basic idea is that you chop up the window into grids, then you specify for each component which cell to start and end in. You can also specify how objects stretch when there is extra room and alignment within cells.

The basic steps to using `GridBagLayout` are:

Set the layout, saving a reference to it.

```
GridBagLayout layout = new GridBagLayout();
setLayout(layout);
```

Allocate a GridBagConstraints object.

```
GridBagConstraints constraints =
  new GridBagConstraints();
```

Set up the GridBagConstraints for component 1.

```
constraints.gridx = x1;
constraints.gridy = y1;
constraints.gridwidth = width1;
constraints.gridheight = height1;
...
```

Add the first component to the window, including constraints.

```
add(component1, constraints);
```

Repeat the last two steps for each remaining component.

The GridBagConstraints Object

In addition to setting the layout in the `Container`, you also need to allocate a `GridBagConstraints` object and specify the constraints for laying out each `Component`. Next, add the `Component` to the `Container` by the `add` method, passing in the `GridBagConstraints` object to control layout of the `Component`. Once you add the `Component` to the `Container`, you can reset the `GridBagConstraints` values, so allocating one `GridBagConstraints` object for each `Component` is not necessary. The `GridBagConstraints` class has many fields, most of which need to be set for every added `Component`. The available fields are described below.

public int gridx

public int gridy

These variables specify the top-left corner of the `Component`.

public int gridwidth

public int gridheight

These variables determine the number of columns and rows the Component occupies. Note that you do not specify the total number of rows or columns in the layout; the `GridBagLayout` determines this number automatically. In fact, you can set `gridwidth` and `gridheight` to `GridBagConstraints.RELATIVE` and then add the components left to right, top to bottom, and let the layout manager try to figure out the widths and heights. If you use this approach, use the value `GridBagConstraints.REMAINDER` for the *last* entry in a row or column. The default value is 1.

public int anchor

If the `fill` field is set to `GridBagConstraints.NONE`, then the `anchor` field determines where the element gets placed. Use `GridBagConstraints.CENTER` to center the component, or `GridBagConstraints.NORTH, GridBagConstraints.NORTHEAST, GridBagConstraints.EAST, GridBagConstraints.SOUTHEAST, GridBagConstraints.SOUTH, GridBagConstraints.SOUTHWEST, GridBagConstraints.WEST`, or `GridBagConstraints.NORTHWEST` to position one side or corner in the allocated box.

public int fill

The size of the row or column is determined by the widest/tallest element contained in the row or column. The `fill` field specifies what to do to an element that is smaller than this size. `GridBagConstraints.NONE` (the default) means not to expand the element at all, `GridBagConstraints.HORIZONTAL` means to expand horizontally but not vertically, `GridBagConstraints.VERTICAL` means to expand vertically but not horizontally, and `GridBagConstraints.BOTH` means to expand the element horizontally and vertically.

public Insets insets

You can use this field to supply an `Insets` object that specifies margins to go on all four sides of the component. The `Insets` class has a single constructor that allows you to specify the four margins around the component:

```
public Insets(int top, int left, int bottom, int right)
```

public int ipadx

public int ipady

These values specify internal padding to add to the component's minimum size. The new size will be the old size plus twice the padding.

public double weightx

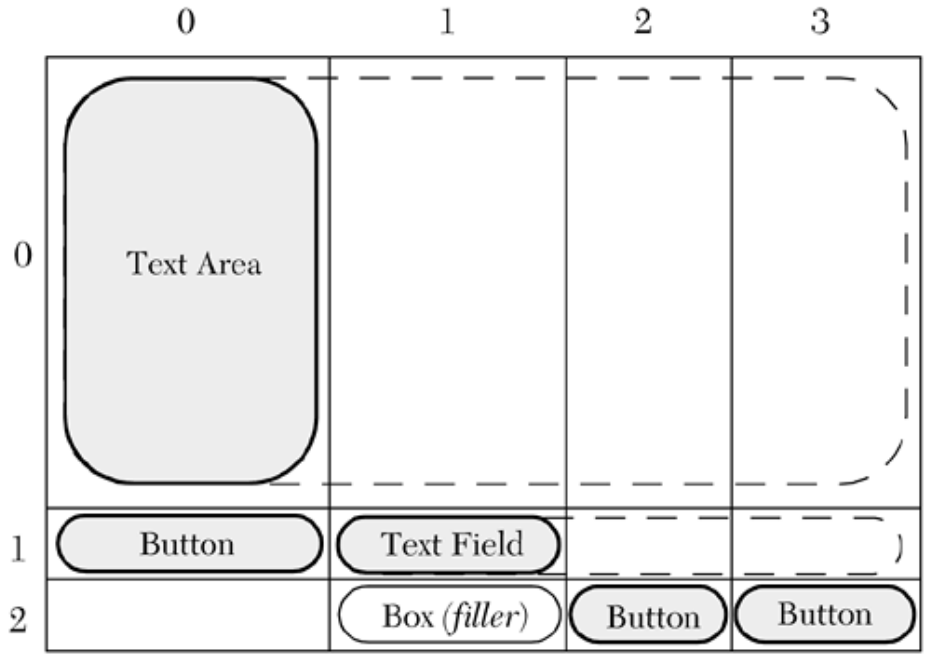
public double weighty

These fields determine how much the `Component` will "stretch" in the x or y direction if space is left over after sizing each column based on the widest object and each row based on the tallest. Specifying 0.0 means that the `Component` will stay at its preferred size. Specifying 100.0 means that the `Component` will share extra space equally with other objects that have a weight of 100.0 specified. Values in between result in prorating.

Example

As an example, consider the an application show in Figure 12-6. The layout contains two input fields (`JTextArea` and `JTextfield`), and three buttons. In the schematic, grid placement of each component is illustrated by a shaded oval, and the cells in which a component can stretch are indicated by a dashed outline. For this design, as the window size increases, the text area should also increase to fill any remaining space. Similarly, the textfield should expand if the width of the window increases. The buttons should remain their original size as the window size changes. Because of the complexity of this layout, a `GridBagLayout` manager is a reasonable choice for controlling the layout of the components.

Figure 12-6. Schematic of `GridBagLayout` showing placement of components and the direction components can expand (stretch).



Listing 12.7 shows the code for setting up the components and constraints for the layout manager. For each component, the `GridBagContraints` are first set, and then the component is added to the `GridBagLayout` panel. In many cases, the `GridBagContraints` instance variables are calculated relative to the previous assignment. In other cases, the instance variables carry over from the previous component assignment and are not reassigned. Behind the scenes, the `GridBagLayout` manager clones a copy of the `GridBagConstraints` and stores the copy in a private `Hashtable`, where the `Component` is the key. Thus, each `Component` always has an associated unique `GridBagContraints` object.

Other than `GridBagLayout`, no other *standard* layout manager will achieve this effect. However, before you leap to the conclusion that `GridBagLayout` is appropriate for this job, let us preview the advice of Section 12.8 (Effective Use of Layout Managers)—consider using multiple nested layouts instead of one huge layout. In fact, in Section 12.8 you'll see how this exact layout can be achieved more easily by use of nested containers. For more information on the Swing components used in this example, see Chapter 14 (Basic Swing).

As in this example, using a *single* layout manager may require the addition of *fillers* (invisible lightweight components) to produce the desired layout. Consider the `Box` added to coordinate (*x*=1, *y*=2). The `Box` is simply a placeholder such that column 1 has at least one component present with a fixed width. Without the invisible `Box`, column 1 would collapse to a width of 0,

because the textfield is stretchable and does not have a fixed horizontal width. The remaining space would be divided equally between columns 2 and 3. As a consequence, the placement of the Ok and Exit buttons would be undesirable, as shown in Figure 12-8. As discussed in Section 12.8 (Effective Use of Layout Managers), you can avoid the need for fillers by mixing or using nested containers.

Figure 12-8. Using `GridBagLayout` without the `Box` as a filler in column 1.

Listing 12.7 `GridBagTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

/** An example demonstrating a GridBagLayout GUI with
 *  input text area and multiple buttons.
 */

public class GridBagTest extends JPanel {
  private JTextArea textArea;
  private JButton bSaveAs, bOk, bExit;
  private JTextField fileField;
  private GridBagConstraints c;

  public GridBagTest() {
    setLayout(new GridBagLayout());
    setBorder(BorderFactory.createEtchedBorder());

    textArea = new JTextArea(12,40);  // 12 rows, 40 cols
    bSaveAs = new JButton("Save As");
    fileField = new JTextField("C:\\Document.txt");
    bOk = new JButton("OK");
    bExit = new JButton("Exit");
```

```
c = new GridBagConstraints();

// Text Area.
c.gridx      = 0;
c.gridy      = 0;
c.gridwidth  = GridBagConstraints.REMAINDER;
c.gridheight = 1;
c.weightx    = 1.0;
c.weighty    = 1.0;
c.fill       = GridBagConstraints.BOTH;
c.insets     = new Insets(2,2,2,2); //t,l,b,r
add(textArea,c);

// Save As Button.
c.gridx      = 0;
c.gridy      = 1;
c.gridwidth  = 1;
c.gridheight = 1;
c.weightx    = 0.0;
c.weighty    = 0.0;
c.fill       = GridBagConstraints.VERTICAL;
add(bSaveAs,c);

// Filename Input (Textfield).
c.gridx      = 1;
c.gridwidth  = GridBagConstraints.REMAINDER;
c.gridheight = 1;
c.weightx    = 1.0;
c.weighty    = 0.0;
c.fill       = GridBagConstraints.BOTH;
add(fileField,c);

// OK Button.
c.gridx      = 2;
c.gridy++;
c.gridwidth  = 1;
c.gridheight = 1;
c.weightx    = 0.0;
c.weighty    = 0.0;
c.fill       = GridBagConstraints.NONE;
add(bOk,c);
// Exit Button.
c.gridx      = 3;
c.gridwidth  = 1;
c.gridheight = 1;
c.weightx    = 0.0;
c.weighty    = 0.0;
c.fill       = GridBagConstraints.NONE;
add(bExit,c);

// Filler so Column 1 has nonzero width.
```

```
    Component filler = Box.createRigidArea(new Dimension(1,1));
    c.gridx      = 1;
    c.weightx    = 1.0;
    add(filler,c);
  }

  public static void main(String[] args) {
    WindowUtilities.setNativeLookAndFeel();
    JFrame frame = new JFrame("GrigBagLayout Test");
    frame.setContentPane(new GridBagTest());
    frame.addWindowListener(new ExitListener());
    frame.pack();
    frame.setVisible(true);
  }
}
```

Figure 12-7. Using `GridBagConstraints` gives you more control, but at a significant cost in code complexity.

GridBagLayout Constructor Options

public GridBagLayout()

`GridBagLayout` has only one constructor. Options are provided through the `GridBagConstraints` object.

Other GridBagLayout Methods

The following are common methods of `GridBagLayout`.

public GridBagConstraints getConstraints(Component c)

This method returns a copy of the `GridBagConstraints` object used to set up the specified component.

public float getLayoutAlignmentX(Container c)

public float getLayoutAlignmentY(Container c)

These methods tell you how the `Container` is to be aligned. For more details, see `getAlignmentX` and `getAlignmentY` in Section 13.2 (The Component Class).

public int[ ][ ] getLayoutDimensions()

This method returns an array containing the dimensions of each row and column in the window.

public Point getLayoutOrigin()

This method returns the location of the top-left corner of the `GridBagLayout` relative to its containing window.

public double[ ][ ] getLayoutWeights

This method retrieves the `weightx` and `weighty` values.

public void setConstraints(Component component, GridBagConstraints constraints)

This method registers the specified constraints with the given component; the method is rarely used because the preferred approach is to simply apply the constraints when adding a `Component` to a `Container`. For example, use

```
container.add(component, constraints)
```

instead of

```
container.add(component);
layout.setConstraints(constraints);
```

12.6 The BoxLayout Manager

`BoxLayout` is a new layout manager introduced in the Swing package. It supports arranging components either in a horizontal row (`BoxLayout.X_AXIS`) or in a vertical column (`BoxLayout.Y_AXIS`). The `BoxLayout` managers lays out the components in the same order in which they were added to the `Container`, left to right for a horizontal layout, and top to bottom for a vertical layout. Resizing the container does not cause the components to relocate.

`BoxLayout` first attempts to arrange the components at their preferred widths (vertical layout) or preferred heights (horizontal layout). For a vertical layout, if all the components are not the same width, `BoxLayout` then attempts to expand all the components to the width of the component with the largest preferred width. If expanding a component is not possible (restricted maximum size), then `BoxLayout` aligns that component horizontally in the container, according to the *x* alignment of the component.

Similarly for a horizontal layout, `BoxLayout` first attempts to arrange the components at their preferred heights. If all the components are not the same height, then `BoxLayout` attempts to expand all the components to the height of the highest component. If expanding the height of a component is not possible, then `BoxLayout` aligns that component vertically in the container, according to the *y* alignment of the component.

Every lightweight Swing component that inherits from `JComponent` can define an alignment value from 0.0f to 1.0f, where 0.0 represents positioning the component closest to the axis origin in the container, and 1.0 represents positioning the component farthest from the axis origin in the container. The `Component` class predefines five alignment values: `Component.LEFT_ALIGNMENT` (0.0), `Component.CENTER_ALIGNMENT` (0.5), `Component.RIGHT_ALIGNMENT` (1.0), `Component.TOP_ALIGNMENT` (0.0), and `Component.BOTTOM_ALIGNMENT` (1.0). Depending on the `BoxLayout` orientation, the alignment of each `JComponent` can be set through the two following methods, where `Xxx` represents the relative position:

```
jcomponent.setAlignmentX(Component.Xxx_ALIGNMENT);
jcomponent.setAlignmentY(Component.Xxx_ALIGNMENT);
```

Most of the Swing components have a default x-axis alignment of center (`Component.CENTER_ALIGNMENT`). `JButton`, `JComboBox`, `JLabel`, and `JMenu` don't follow this general rule. These components have x-axis alignment of left (`Component.LEFT_ALIGNMENT`).

A simple example of a vertical `BoxLayout` (`BoxLayout.Y_AXIS`) is shown in Listing 12.8. The layout contains a label at the top with a default left alignment and three buttons that, when individually selected, will call a method to accordingly change the alignment of all three buttons and then revalidate the `Container`. The left default alignment of the label does not change.

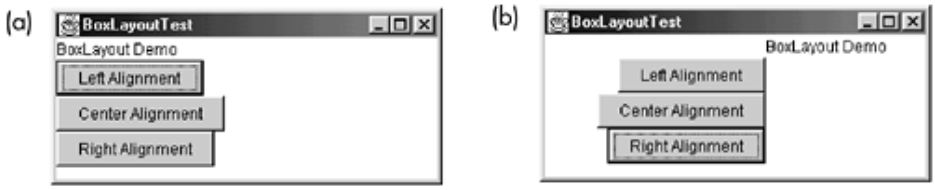
Listing 12.8 `BoxLayoutTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** An example of BoxLayout. */

public class BoxLayoutTest extends JPanel
                           implements ActionListener{
  BoxLayout layout;
  JButton topButton, middleButton, bottomButton;

  public BoxLayoutTest() {
    layout = new BoxLayout(this, BoxLayout.Y_AXIS);
    setLayout(layout);

    JLabel label = new JLabel("BoxLayout Demo");

    topButton = new JButton("Left Alignment");
    middleButton = new JButton("Center Alignment");
    bottomButton = new JButton("Right Alignment");
    topButton.addActionListener(this);
    middleButton.addActionListener(this);
    bottomButton.addActionListener(this);
    add(label);
    add(topButton);
    add(middleButton);
    add(bottomButton);
```

```
    setBackground(Color.white);
  }

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == topButton) {
      refresh(Component.LEFT_ALIGNMENT);
    } else if (event.getSource() == middleButton) {
      refresh(Component.CENTER_ALIGNMENT);
    } else if (event.getSource() == bottomButton) {
      refresh(Component.RIGHT_ALIGNMENT);
    }
  }

  private void refresh(float alignment){
    topButton.setAlignmentX(alignment);
    middleButton.setAlignmentX(alignment);
    bottomButton.setAlignmentX(alignment);
    revalidate();
    System.out.println("x: "+layout.getLayoutAlignmentX(this));
  }

  public static void main(String[] args) {
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(new BoxLayoutTest(), 300, 135,
                                 "BoxLayoutTest");
  }
}
```

Figure 12-9(a) and 12-9(b) show the `BoxLayout` after the Left Alignment button and Right Alignment button, respectively, are selected. This example illustrates the problem when the components have differing alignment values.

Figure 12-9. (a) `BoxLayout` where all components have a 0.0 (left) alignment. (b) `BoxLayout` where the label has a 0.0 alignment and the buttons have a 1.0 (right) alignment.



In Figure 12-9(a), all the components have a left alignment and are correctly aligned to the *x*-axis origin. But in Figure 12-9(b), the three buttons have a *right* alignment, while the label maintains the original *left* alignment. As shown, if the components have different alignment values, the alignment of the components is no longer simply relative to the container but also relative to the other components in the container. In this situation, the alignment value (0.0–1.0) represents the portion of the component that is to the left of a common "component" axis (the component is always in contact with the component axis). Thus, 0% of the label (alignment 0.0) is to the left of the component axis, and 100% of the buttons (alignment 1.0) are to the left of the component axis. The location of the component axis relative to the container is based on a prorated calculation of the individual component alignments and widths. In this latter case, the component axis alignment

(or simply, the alignment of the `Container`) is 0.581. To avoid the unexpected alignment in Figure 12-9(b), we recommend that you always set the alignment value for each component, and if appropriate, use the same alignment value for all the components in your `BoxLayout` design.

For a vertical `BoxLayout`, the default behavior is to size the width of all components to the width of the widest component. However, if a component has an unbounded maximum width, for example, `Integer.MAX_VALUE`, then that component will always be as wide as the `Container`. An unusual circumstance is where all the components have an unbounded width, but at least one of them has a *strict* left (0.0) or right (1.0) alignment. For this situation, the width of the components with a strict alignment will always be *less* than the width of the `Container`, while the other components will be as wide as the `Container`. In practice, nearly all components report the same maximum size as the preferred size and do not have an unbounded width, so this behavior of `BoxLayout` goes unnoticed. You can change the maximum size of a `Component` through

```
component.setMaximumSize(new Dimension(width, height));
```

A similar analogy holds for component heights in a horizontal layout.

BoxLayout Constructor Options

The `BoxLayout` manager has only one constructor.

public BoxLayout(Container container, int axis)

The first parameter in the `BoxLayout` constructor is a reference to the `Container` that the `BoxLayout` is managing. The `Container` is referenced internally as a private variable and checked when calling other `BoxLayout` methods. Thus, each `BoxLayout` manager can only be assigned to one `Container`. The `axis` parameter specifies the direction of the layout, either `BoxLayout.X_AXIS` for a left-to-right (horizontal) layout or `BoxLayout.Y_AXIS` for a top-to-bottom (vertical) layout.

Core Alert

Unlike the other standard layout managers, the `BoxLayout` manager cannot be shared with more than one `Container`.

Other BoxLayout Methods

The following are the most common methods of the `BoxLayout` class.

public Dimension preferredLayoutSize(Container container)

public Dimension minimumLayoutSize(Container container)

public Dimension maximumLayoutSize(Container container)

These methods, respectively, return the preferred, minimum, and maximum `Dimension` necessary to lay out the `Container`, given the contained components.

public float getLayoutAlignmentX(Container container)

public float getLayoutAlignmentY(Container container)

These methods return the desired overall *x*- or *y*-axis alignment, respectively, for the `Container` (which is a `Component`), relative to *other* components in the window. The overall alignment of the `Container` is based on the required alignment of each internal `Component` in the `Container`. The alignment can range from 0.0f to 1.0f, inclusively. See Section 13.2 (The Component Class) for additional details on alignment.

public void layoutContainer(Container container)

This method positions and sizes all the components in the Container by calling setBounds on each Component.

public void invalidateLayout(container container)

This method discards any cached information for laying out the components.

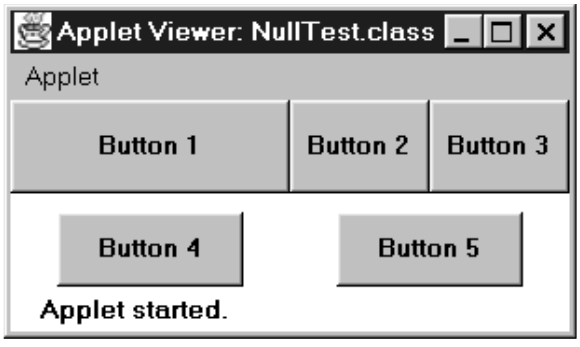
Note that Swing also provides a `Box` container that uses a `BoxLayout` manager. In the `Box` constructor you can specify the axis orientation as `BoxLayout.X_AXIS` for a horizontal layout or `BoxLayout.Y_AXIS` for a vertical layout. The `Box` class also provides multiple static methods for creating invisible filler components. See Section 12.8 (Effective Use of Layout Managers) for details.

12.7 Turning Off the Layout Manager

Layout managers are supposed to make your life easier. If you find they're getting in the way, just turn them off. This action is no sin, although with a little experience you'll find that you want to do this less often than you might think at first. If the layout is set to `null`, then components can be (in fact, *must* be) sized and positioned by hand, with either `setSize(width, height)` and `setLocation(left, top)` or `setBounds(left, top, width, height)`. Applets are not resizable, but for `Frame` and `JFrame` you could override `setSize` and `setBounds` to call `super.setSize` and `super.setBounds`, respectively, and reposition the components.

Although this technique sounds tedious, if the window with the layout manager turned off contains only a few components (possibly other windows with non-`null` layout managers), this approach is quite tractable. Listing 12.9 gives an example, with the result shown in Figure 12-10. As we'll see in Section 12.8 (Effective Use of Layout Managers), using nested containers and basing dimensions on the current window size rather than on absolute pixels makes this approach significantly more flexible while requiring very little extra work.

Figure 12-10. You can position elements by hand if you set the layout manager to `null`.



Listing 12.9 `NullTest.java`

```
import java.applet.Applet;
import java.awt.*;

/** Layout managers are intended to help you, but there
 *  is no law saying you <B>have</B> to use them.
 *  Set the layout to null to turn them off.
 */

public class NullTest extends Applet {
  public void init() {
    setLayout(null);
    Button b1 = new Button("Button 1");
    Button b2 = new Button("Button 2");
    Button b3 = new Button("Button 3");
    Button b4 = new Button("Button 4");
    Button b5 = new Button("Button 5");
    b1.setBounds(0, 0, 150, 50);
    b2.setBounds(150, 0, 75, 50);
    b3.setBounds(225, 0, 75, 50);
    b4.setBounds(25, 60, 100, 40);
    b5.setBounds(175, 60, 100, 40);
    add(b1);
    add(b2);
    add(b3);
    add(b4);
    add(b5);
  }
}
```

12.8 Effective Use of Layout Managers

The idea of a layout manager is a good one. In principle, a layout manager relieves the developer of the requirement to specify the sizes and positions of each and every component and simplifies the development of many interfaces. In practice, however, many developers feel that the layout managers are hindering them more than helping them. Although some layout managers such as `GridBagLayout` require more effort than they should, a few basic strategies for harnessing the power of layout managers will significantly simplify the UI layout process.

Use nested containers. No law states that you have to use a single layout manager for everything. Divide and conquer!

Turn off the layout manager for some containers. In a pinch, don't be afraid to position a few things by hand.

Use a custom layout manager. Don't like the standard layout managers? Write your own.

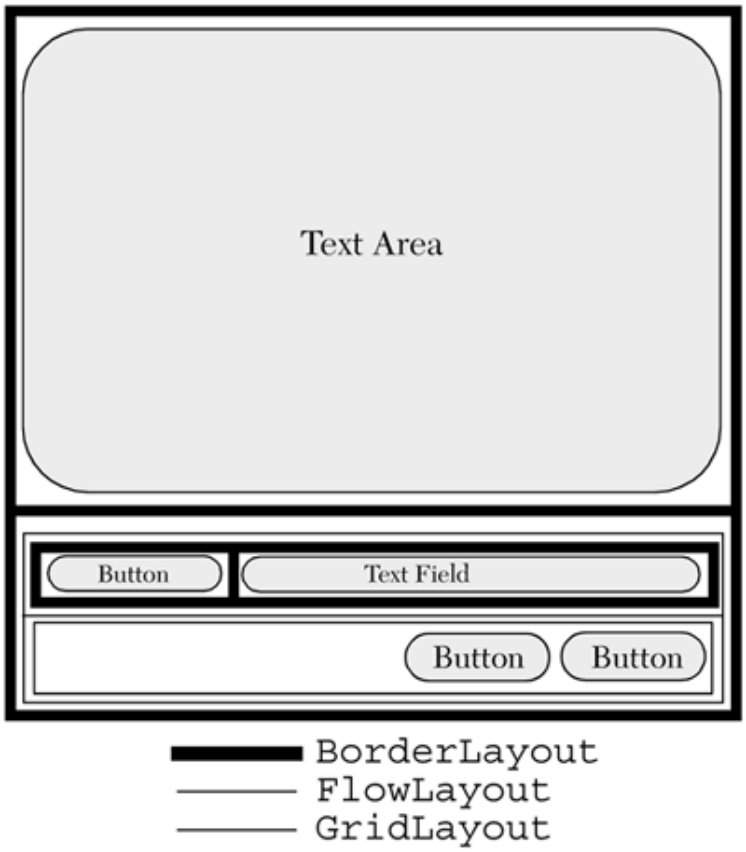
Adjust the empty space around components. If your layout is almost perfect except for some empty space, there are a few small tweaks you can use to fix up those last details.

Use Nested Containers

Rather than struggling to fit your design into a single layout, try dividing the design into rectangular sections. Let each section be a panel with a separate layout manager. In Section 12.5 (GridBagLayout), we showed you a layout incorporating various input fields and buttons. The original layout was performed with a single `GridBagLayout` manager, but as Listing 12.10 shows, you could do the layout more easily by creating separate panels for the various regions and applying a different layout manager to each panel.

A schematic of the nested container layout is shown in Figure 12-11. The key idea when nesting different layouts is to remember how the different standard layout managers respect the preferred size of the contained components. Specifically,

Figure 12-11. Schematic of nested containers showing placement of components and implied panels. The layout manager for each panel is indicated by the key.



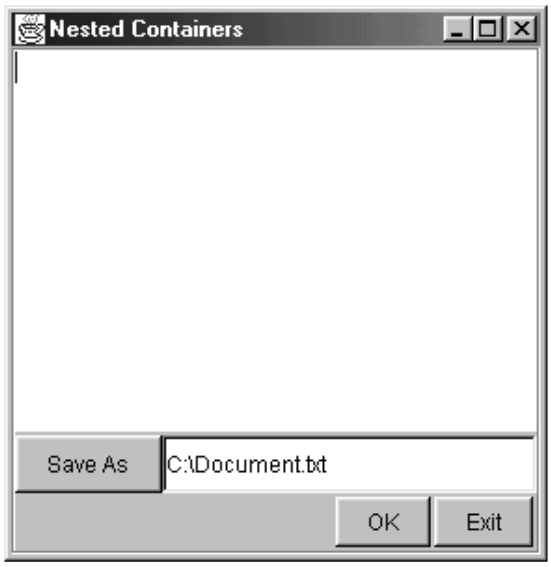
- `FlowLayout`— Respects the preferred size of the components.
- `GridLayout`— Does not respect the preferred size of the components.
- `BorderLayout`— `North` and `South` respect the preferred height, `East` and `West` respect the preferred width, `Center` respects neither the width nor height of the component.

Based on these behaviors, the nested layout shown in Figure 12-11 falls out naturally. The text area should expand both horizontally and vertically to fill all remaining space; thus, the *preferred* width and height need not be respected. The button components at the bottom should maintain a fixed height, and the textfield should be allowed to expand horizontally. Thus, a `BorderLayout` manager is a natural choice for the first `Container`, where the `EAST`, `NORTH`, and `WEST`

locations are empty.

The bottom area geometrically break up into two rows, so, a `JPanel` with a new `GridLayout` manager defining two rows and one column is added to the `SOUTH` location. To handle the input field and multiple buttons, a `JPanel` is added to each row of the `GridLayout`. Each `JPanel` is assigned a new layout manager to better control positioning of the components. To determine the best layout manager for each panel, examine the requirements of the contained components. The preferred size of the Save As `JButton` should be respected, whereas the input field, `JTextField`, should expand horizontally with the window. Again, a `BorderLayout` is appropriate, where the `NORTH`, `EAST`, and `SOUTH` locations are empty. Lastly, the Ok `JButton` and Exit `JButton` should stay their preferred size but always be located in the far-right location of the `JPanel`. The default layout manager of a `JPanel`, `FlowLayout`, respects the size of all components but, by default, centers the components. All that is needed is to reassign the `FlowLayout` manager with a component alignment of `FlowLayout.RIGHT`. Ta da! Of course, other nested container layouts are possible, but this example clearly shows that a nested container approach significantly simplifies the code compared to that for a single `GridBagLayout` manager, as in Listing 12.7. The final result is shown in Figure 12-12.

Figure 12-12. Nested containers generally simplify complex layouts.



Listing 12.10 `NestedLayout.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
/** An example demonstrating the use of nested containers
 *  to lay out the components. See GridBagTest.java for
 *  implementation by a single layout manager, GridBagLayout.
 */

public class NestedLayout extends JPanel {

  private JTextArea textArea;
```

```
  private JButton bSaveAs, bOk, bExit;
  private JTextField fileField;

  public NestedLayout() {

    setLayout(new BorderLayout(2,2));
    setBorder(BorderFactory.createEtchedBorder());

    textArea = new JTextArea(12,40);  // 12 rows, 40 cols
    bSaveAs = new JButton("Save As");
    fileField = new JTextField("C:\\Document.txt");
    bOk = new JButton("OK");
    bExit = new JButton("Exit");

    add(textArea,BorderLayout.CENTER);

    // Set up buttons and textfield in bottom panel.
    JPanel bottomPanel = new JPanel();
    bottomPanel.setLayout(new GridLayout(2,1));

    JPanel subPanel1 = new JPanel();
    JPanel subPanel2 = new JPanel();
    subPanel1.setLayout(new BorderLayout());
    subPanel2.setLayout(new FlowLayout(FlowLayout.RIGHT,2,2));

    subPanel1.add(bSaveAs,BorderLayout.WEST);
    subPanel1.add(fileField,BorderLayout.CENTER);
    subPanel2.add(bOk);
    subPanel2.add(bExit);

    bottomPanel.add(subPanel1);
    bottomPanel.add(subPanel2);

    add(bottomPanel,BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    WindowUtilities.setNativeLookAndFeel();
    JFrame frame = new JFrame("Nested Containers");
    frame.setContentPane(new NestedLayout());
    frame.addWindowListener(new ExitListener());
    frame.pack();
    frame.setVisible(true);
  }
}
```

Turn Off the Layout Manager for Some Containers

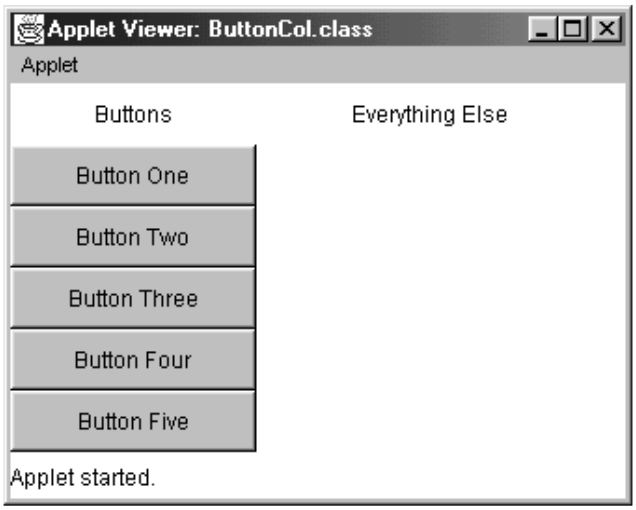
Positioning components individually is not always particularly difficult, especially if combined with the use of nested windows. For instance, suppose that you wanted to arrange an applet with a column of buttons down the left side, with the remainder of the space reserved for drawing space (with a `Label` as a title). Further suppose that you wanted the button column to take up exactly 40% of the width of the applet, rather than a width independent of the `Container` size (enough

space for the widest button). You can implement the design by allocating two `Panels` but positioning them by using `setBounds` after turning off the layout manager for the overall applet. Because each `Panel` still has a layout manager, components inside of them do *not* need to be positioned manually.

Listing 12.11 `ButtonCol.java`

```
import java.applet.Applet;
import java.awt.*;

/** An example of a layout performed manually. The top-level
 *  panels are positioned by hand, after you determine the size
 *  of the applet. Since applets can't be resized in most
 *  browsers, setting the size once when the applet is created
 *  is sufficient.
 */

public class ButtonCol extends Applet {
  public void init() {
    setLayout(null);
    int width1 = getSize().width*4/10,
        width2 = getSize().width - width1,
        height = getSize().height;
    Panel buttonPanel = new Panel();
    buttonPanel.setBounds(0, 0, width1, height);
    buttonPanel.setLayout(new GridLayout(6, 1));
    buttonPanel.add(new Label("Buttons", Label.CENTER));
    buttonPanel.add(new Button("Button One"));
    buttonPanel.add(new Button("Button Two"));
    buttonPanel.add(new Button("Button Three"));
    buttonPanel.add(new Button("Button Four"));
    buttonPanel.add(new Button("Button Five"));
    add(buttonPanel);
    Panel everythingElse = new Panel();
    everythingElse.setBounds(width1+1, 0, width2, height);
    everythingElse.add(new Label("Everything Else"));
    add(everythingElse);
  }
}
```

Figure 12-13. Manually position top-level windows, not individual components.



Adjust the Empty Space Around Components

Change the space allocated by the layout manager. All the standard AWT layout managers except for `GridBagLayout` have a constructor option that lets you specify the amount of space that will be reserved between components. With `GridBagLayout`, you can specify this option by using the `insets` or `ipadx` and `ipady` fields of the `GridBagConstraints` object.

Override insets in the Container. `Insets` are designed to be empty margins around the inside of a `Container`. All built-in layout managers respect them, as should any custom layout managers. `Insets` allows for borders around components. Unfortunately, however, a `Container` does not provide a way to change its insets. So, a subclass of `Container` (probably a `Panel` subclass) must be created to override the `getInsets` method.

Use a Canvas or a Box as an invisible spacer. For AWT layouts, use a `Canvas` that does not draw or handle mouse events as an "empty" component for spacing. A `Canvas` works as an invisible spacer in windows using `FlowLayout`, `GridLayout`, `GridBagLayout`, `BoxLayout` or in the noncenter region of a window using `BorderLayout`.

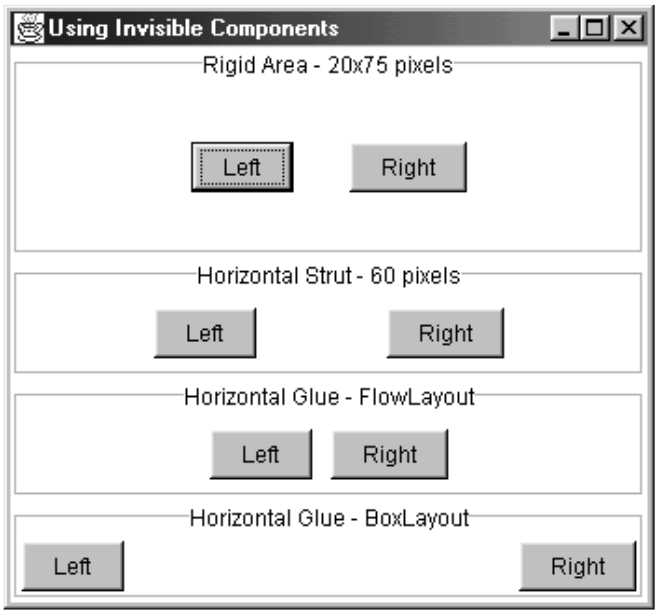
For Swing layouts, add a `Box` as an invisible spacer to improve positioning of components. The `Box` class provides three types of spacers:

- Rigid area— A two-dimensional invisible component (`Box.createRigidArea`) with a fixed width and height.
- Strut— A one-dimensional invisible component that can be either a *horizontal* strut with fixed width and (`Box.CreateHorizontalStrut`) zero height, or a *vertical* strut with zero width and (`Box.CreateHorizontalStrut`) fixed height.
- Glue— A one- (`Box.createHorizontalGlue`, `Box.createVerticalGlue`) or two-dimensional (`Box.createGlue`) invisible component that will expand to fill all remaining space.

Listing 12.12 illustrates the use of a rigid area, a horizontal strut, and horizontal glue. Four separate `JPanel`s (default `FlowLayout` manager) are created, each containing a left and right

button. As shown in Figure 12-14, the first panel incorporates an invisible rigid area (20 x 75 pixels) between the two buttons. The second panel incorporates a horizontal strut, 60 pixels wide with no height, to separate the buttons. The third panel attempts to use a `Box` component as glue to fill in the space between the buttons, but fails, because `FlowLayout` does not respect the *maximum* size of the `Box`. `BoxLayout` is the only manager that respects the desired maximum size of a component. The fourth panel, using a `BoxLayout`, shows the correct effect of horizontal glue.

Figure 12-14. An invisible `Box` can improve the layout of the components.



Core Alert

Only apply "glue" to layout managers that respect the maximum size of a `Component`. The use of a glue component is correctly supported in `BoxLayout`.

Listing 12.12 `InvisibleComponentTest.java`

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class InvisibleComponentTest extends JPanel {
  Component spacer;

  public InvisibleComponentTest() {
    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

    // Place a rigid invisible component 25 pixels wide and
    // 75 pixels tall between the two buttons
    JPanel p1= new JPanel();
    spacer = Box.createRigidArea(new Dimension(20,75));
    setUpPanel(p1, "Rigid Area - 20x75 pixels", spacer);

    // Separate two buttons by a 60-pixel horizontal strut
```

```
    JPanel p2= new JPanel();
    spacer = Box.createHorizontalStrut(60);
    setUpPanel(p2, "Horizontal Strut - 60 pixels", spacer);

    // Horizontal glue in FlowLayout - not useful
    JPanel p3= new JPanel();
    spacer = Box.createHorizontalGlue();
    setUpPanel(p3, "Horizontal Glue - FlowLayout", spacer);

    // Add glue to fill all remaining horizontal space between
    // the two buttons. Glue not supported by default FlowLayout
    // of JPanel.  Change layout of JPanel to BoxLayout.
    JPanel p4= new JPanel();
    p4.setLayout(new BoxLayout(p4,BoxLayout.X_AXIS));
    spacer = Box.createHorizontalGlue();
    setUpPanel(p4, "Horizontal Glue - BoxLayout", spacer);

    add(p1);
    add(p2);
    add(p3);
    add(p4);
  }

  // Helper to set the border and add components
  private void setUpPanel(JPanel p, String title,
                          Component spacer) {
    p.setBorder(BorderFactory.createTitledBorder(
                      BorderFactory.createEtchedBorder(),title,
                      TitledBorder.TOP,TitledBorder.CENTER));
    p.setBackground(Color.white);
    p.add(new JButton("Left"));
    p.add(spacer);
    p.add(new JButton("Right"));
  }

  public static void main(String[] args) {
    String title = "Using Invisible Components";
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(new InvisibleComponentTest(),
                                 350, 325, title);
  }
}
```

Box Utility Methods

The `Box` class provides the following `static` methods for creating invisible components:

public static Component createRigidArea(Dimension size)

This method creates an invisible `Component` that maintains a fixed `Dimension` (width and height).

public static Component createHorizontalStrut(int width)

public static Component createVerticalStrut(int height)

The first method creates an invisible `Component` of fixed width and zero height. The second method creates an invisible `Component` of fixed height and zero width.

public static Component createHorizontalGlue()

public static Component createVerticalGlue()

public static Component createGlue()

The first two methods create an invisible glue `Component` that can expand horizontally or vertically, respectively, to fill all remaining space. The third method creates a `Component` that can expand in both directions. A `Box` object achieves the glue effect by expressing a maximum size of `Short.MAX_VALUE`. If more than one glue component is present in a `BoxLayout`, the remaining space is divided equally between the glue components.

12.9 Summary

Layout managers help you position components in the window. They are particularly helpful when you resize the window, add components, or move the program among operating systems. There are five traditional layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, and `GridBagLayout`. In Java 2, the `BoxLayout` manager was added for additional flexibility. If you don't like any of those, you can always turn off the layout manager, but clever use of nested containers often makes turn-off unnecessary.

Okay, so you've got windows galore. You're a layout manager pro. But putting something *in* the windows would be nice. Read on: Chapter 13 covers buttons, check boxes, radio buttons, scrollbars, and other GUI controls.

CONTENTS