Custom Painting for Swing components

In these notes we shall be considering the following:

1 - How to set the look and feel of an application
2 - How to set layout manager of a container
3 - Painting
4 - How to use borders

These notes are a selection of a set of notes developed by javasoft and which can be found at [http://java.sun.com/docs/books/tutorial/uiswing/index.html](http://java.sun.com/docs/books/tutorial/uiswing/index.html). Downloadable source code of the examples are also available from the same site.

1 Setting the Look and Feel

Most programs do not specify the Look and Feel of the application. When a program does not set its look and feel, the Swing UI manager must figure out which look and feel to use. It first checks whether the user has specified a preferred look and feel. If so, it attempts to use that. If not, or if the user's choice is not valid, then the UI manager chooses the Java Look & Feel.

To specify a look and feel, use the `UIManager.setLookAndFeel` method. For example, the bold code in the following snippet makes the program use the Java Look & Feel:

```java
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }
    new SwingApplication(); //Create and show the GUI.
}
```

The argument to `setLookAndFeel` is the fully qualified name of the appropriate subclass of `LookAndFeel`. To specify the Java Look & Feel, we used the `getCrossPlatformLookAndFeelClassName` method. If you want to specify the native look and feel for whatever platform the user runs the program on, use the `getSystemLookAndFeelClassName` method. To specify a particular UI, you can use the actual class name. For example, if you design a program to look best with the Windows Look & Feel, you can use this code to set the look and feel:

```java
UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
);
```
Here are some of the arguments you can use for `setLookAndFeel`:

`UIManager.getCrossPlatformLookAndFeelClassName()`
Returns the string for the one look-and-feel guaranteed to work -- the Java Look & Feel.

`UIManager.getSystemLookAndFeelClassName()`
Specifies the look and feel for the current platform. On Win32 platforms, this specifies the Windows Look & Feel. On Mac OS platforms, this specifies the Mac OS Look & Feel. On Sun platforms, it specifies the CDE/Motif Look & Feel.

"javax.swing.plaf.metal.MetalLookAndFeel"
Specifies the Java Look & Feel. (The codename for this look and feel was Metal.) This string is the value returned by the `getCrossPlatformLookAndFeelClassName` method.

"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
Specifies the Windows Look & Feel. Currently, you can use this look and feel only on Win32 systems.

"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
Specifies the CDE/Motif Look & Feel. This look and feel can be used on any platform.

"javax.swing.plaf.mac.MacLookAndFeel"
Specifies the Mac OS Look & Feel, which can be used only on Mac OS platforms.

You aren't limited to the preceding arguments. You can specify the name for any look and feel that is in your program's class path.

1.1 How the UI Manager Chooses the Look and Feel

Here are the look-and-feel determination steps that occur when the UI manager first initializes itself:

1. If the program sets the look and feel before any components are created, the UI manager tries to create an instance of the specified look-and-feel class. If successful, all components use that look and feel.

2. If the program hasn't successfully specified a look and feel, then before the first component's UI is created, the UI manager tests whether the user specified a look and feel in a file named `swing.properties`. It looks for the file in the `lib` directory of the Java release. For example, if you're using the Java interpreter in `javaHomeDirectory\bin`, then the `swing.properties` file (if it exists) is in
If the user specified a look and feel, then again the UI manager tries to instantiate the specified class. Here is an example of the contents of a `swing.properties` file:

3. # Swing properties
4.
5. swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
6. If neither the program nor the user successfully specifies a look and feel, then the program uses the Java Look & Feel.

### 1.2 Changing the Look and Feel After Startup

You can change the look and feel with `setLookAndFeel` even after the program's GUI is visible. To make existing components reflect the new look and feel, invoke the `SwingUtilities.updateComponentTreeUI` method once per top-level container. Then you might wish to resize each top-level container to reflect the new sizes of its contained components. For example:

```java
UIManager.setLookAndFeel(lnfName);
SwingUtilities.updateComponentTreeUI(frame);
frame.pack();
```

### 2 Using Layout Managers

Every container, by default, has a layout manager -- an object that implements the `LayoutManager` interface. If a container's default layout manager doesn't suit your needs, you can easily replace it with another one. The Java platform supplies layout managers that range from the very simple (`FlowLayout` and `GridLayout`) to the special purpose (`BorderLayout` and `CardLayout`) to the very flexible (`GridBagLayout` and `BoxLayout`).

This section gives you an overview of some layout managers that the Java platform provides, gives you some general rules for using layout managers, and then tells you how to use each of the provided layout managers. It also points to examples of using each layout manager.

#### 2.1 How to Use BorderLayout

`BorderLayout` is the default layout manager for every content pane (the content pane is the main container in all frames, applets, and dialogs.) A `BorderLayout` has five areas available to hold components: north, south, east, west, and center. All extra space is placed in the center area. Here's an gui that puts one button in each area:
2.2 How to Use BoxLayout

The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes, and also lets you align components.

Here's a GUI that uses a BoxLayout to put a bunch of buttons in a centered column:

![BoxLayout Example]

2.3 How to Use CardLayout

The CardLayout class lets you implement an area that contains different components at different times. Tabbed panes are intermediate Swing containers that provide similar functionality, but with a pre-defined GUI. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. Here's an GUI that uses a combo box and CardLayout in this way:

![CardLayout Example]
2.4 How to Use FlowLayout

FlowLayout is the default layout manager for every JPanel. It simply lays out components from left to right, starting new rows if necessary. Both panels in the CardLayout gui above use FlowLayout. Here's another example of an applet that uses a FlowLayout:

![FlowLayout Example](image1.png)

2.5 How to Use GridLayout

GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns. Here's a gui that uses a GridLayout to control the display of five buttons:

![GridLayout Example](image2.png)

2.6 How to Use GridBagLayout

GridBagLayout is the most sophisticated, flexible layout manager the Java platform provides. It aligns components by placing them within a grid of cells, allowing some components to span more than one cell. The rows in the grid aren't necessarily all the same height; similarly, grid columns can have different widths. Here's a gui that uses a GridBagLayout to manage five buttons:

![GridBagLayout Example](image3.png)
3 Painting

In this section we shall be considering topics related to custom painting. This section will provide info that will help you if your components don't seem to be painting themselves correctly.

3.1 How Painting Works

When a Swing GUI needs to paint itself -- whether for the first time, in response to becoming unhidden, or because it needs to reflect a change in the program's state -- it starts with the highest component that needs to be repainted and works its way down the containment hierarchy. This process is orchestrated by the AWT painting system, and made more efficient and smooth by the Swing repaint manager and double-buffering code.

Swing components generally repaint themselves whenever necessary. When you invoke the `setText` method on a component, for example, the component should automatically repaint itself and, if appropriate, resize itself. If it doesn't, it's a bug. The workaround is to invoke the `repaint` method on the component to request that the component be scheduled for painting. If the component's size or position needs to change but doesn't do so automatically, you should invoke `revalidate` upon the component before invoking `repaint`.

Like event-handling code, painting code executes on the event-dispatching thread. While an event is being handled, no painting will occur. Similarly, if a painting operation takes a long time, no events will be handled during that time. Programs should paint only when the painting system tells them to because each occurrence of a component painting itself must execute without interruption. Otherwise, unpredictable results could occur, such as a button being painted as half pressed and half unpressed.

For smoothness, Swing painting is *double-buffered* by default -- performed to an offscreen buffer and then flushed to the screen once finished. It might slightly help performance if you make a Swing component opaque, so that the Swing painting system can know not to paint anything behind the component. To make a Swing component opaque, invoke `setOpaque(true)` on the component.

Although their available painting area is always rectangular, non-opaque Swing components can appear to be any shape. A button, for instance, might display itself by painting a filled octagon. The component behind the button (its container, most likely) would then be visible, showing through at the corners of the button's bounds. The button would have to include special hit detection code to avoid acting pressed if the user happens to click on its corners.
3.2 An Example of Painting

To illustrate painting, we'll use the `SwingApplication` program. Here is SwingApplication's GUI:

![SwingApplication GUI]

Here, again, is its containment hierarchy:

```
JFrame
  ...
  content pane
    JPanel
      JButton
      JLabel
```

When the GUI for `SwingApplication` is painted, here's what happens:

1. The top-level container, `JFrame`, paints itself.
2. The content pane first paints its background, which is a solid gray rectangle. It then tells the `JPanel` to paint itself. The content pane's background rectangle doesn't actually appear in the finished GUI because the content pane is completely obscured by the `JPanel`.

   **Note:** It's important that the content pane be opaque. Otherwise, messy repaints will result. Because the `JPanel` is opaque, we could make it the content pane (by substituting `setContentPane` for the existing code `getContentPane().add`). This would slightly simplify the containment hierarchy and painting by removing an unnecessary container.

3. The `JPanel` first paints its background, a solid gray rectangle. Next, it paints its border. The border is an `EmptyBorder`, which has no effect except for increasing the `JPanel`'s size by reserving some space at the edge of the panel. Finally, the panel asks its children to paint themselves.
4. To paint itself, the `JButton` paints its background rectangle, if necessary, and then paints the text that it contains. If the button has the keyboard focus, meaning that any typing goes directly to the button for processing, then the button does some look-and-feel-specific painting to make clear that it has the focus.

5. To paint itself, the `JLabel` paints its text.

In this way, each component paints itself before any of the components it contains. This ensures that the background of a `JPanel`, for example, is visible only where it isn't covered by painting performed by one of the components it contains. The following figure illustrates the order in which each component that inherits from `JComponent` paints itself:

1. background (if opaque)
2. custom painting (if any)
3. border (if any)
4. children (if any)

![Custom Painting Diagram]

### 3.3 Overview of Custom Painting

Before you implement a component that performs custom painting, first make sure that you really need to do so. You might be able to use the text and image capabilities of `labels`, `buttons`, or `text components` instead. And remember, you can use `borders` to customize the outside edges of a component.

If you really need to perform custom painting, then you need to decide which superclass to use. We recommend that you extend either `JPanel` or a more specialized Swing component class. For example, if you're creating a custom button class, you should probably implement it by extending a button class such as `JButton` or `JToggleButton`. That way you'll inherit the state management provided by those classes. If you're creating a component that paints on top of an image, you might want to create a `JLabel` subclass. On the other hand, if you're implementing a component that generates and displays a graph on top of a blank or transparent background, then you might want to use a `JPanel` subclass.

When implementing custom painting code, keep two things in mind:

- Your custom painting code belongs in a method named `paintComponent`.
- You can -- and probably should -- use a border to paint the outside edges of your component.
3.4 An Example of Custom Painting

The following code gives an example of custom painting. It shows an image twice, once at its natural size and once very wide.

```java
class ImagePanel extends JPanel {
    ...
    public void paintComponent(Graphics g) {
        super.paintComponent(g); //paint background
        //Draw image at its natural size first.
        g.drawImage(image, 0, 0, this); //85x62 image
        //Now draw the image scaled.
        g.drawImage(image, 90, 0, 300, 62, this);
    }
}
```

Here is the result:

![Image Result]

The example demonstrates a few rules that apply to all components that perform custom painting:

- The painting code does something that no standard Swing component does. If we just wanted to display the figure once, at its natural size, we would have used a `JLabel` object instead of the custom component.
- The custom component is a `JPanel` subclass. This is a common superclass for custom components.
- All the custom painting code is in a method called `paintComponent`.
- Before performing any custom painting, the component paints its background by invoking `super.paintComponent`. If we remove that call, either our custom painting code must paint the component's background or we must invoke `setOpaque(false)` on the component. Doing the latter would inform the Swing painting system that the components behind the non-opaque component might be visible, and thus should be painted.

One thing this component does not do is take borders into account. Not only does it not use a border, but it also doesn't adjust its painting coordinates to take a border into account. A production-quality component would adjust to borders as described in the next subsection.
3.5 The Coordinate System

Each component has its own integer coordinate system, ranging from (0, 0) to (width - 1, height - 1), with each unit representing the size of one pixel. As the following figure shows, the upper left corner of a component's painting area is (0, 0). The X coordinate increases to the right, and the Y coordinate increases downward.

When painting a component, you must take into account not only the component's size but also the size of the component's border, if any. For example, a border that paints a one-pixel line around a component changes the top leftmost corner from (0,0) to (1,1) and reduces the width and the height of the painting area by two pixels each (one pixel per side). The following figure demonstrates this:

You get the width and height of a component using its `getWidth` and `getHeight` methods. To determine the border size, use the `getInsets` method. Here is some code that a component might use to determine the width and height available for custom painting:

```java
public void paintComponent(Graphics g) {
  ...
  Insets insets = getInsets();
  int currentWidth = getWidth() - insets.left - insets.right;
  int currentHeight = getHeight() - insets.top - insets.bottom;
  ...
  /* First painting occurs at (x,y), where x is at least
   insets.left, and y is at least insets.height. */ ...
}```
To familiarize yourself with the coordinate system, you can play with the following applet. Wherever you click on or inside the framed area, a dot is painted, and the label below lists the click's coordinates. The dot is obscured if you click on the border because the component's border is painted after the component performs its custom painting. If we didn't want this effect, an easy solution would be to move the border from the component into a new JPanel object that contains the component.

3.6 Arguments to the repaint Method
Remember that calling a component's repaint method requests that the component be scheduled to paint itself. When the painting system is unable to keep up with the pace of repaint requests, it might combine multiple requests into a single paint request to the component.

The repaint method has two useful forms:

void repaint()
Requests that the entire component be repainted.

void repaint(int, int, int, int)
Requests that only the specified part of the component be repainted. The arguments specify first the X and Y coordinates at the upper left of the area to be repainted, and then the area's width and height.

Although using the four-argument form of repaint method often isn't practical, it can help painting performance significantly. The program in the following picture uses the four-argument repaint method when requesting frequent repaints to display the user's current selection area. Doing so avoids repainting the parts of the component that haven't changed since the last painting operation.
Here is the code that calculates the area to be repainted and then paints it:

```java
class SelectionArea extends JLabel {
    
    public SelectionArea(ImageIcon image, ...) {
        super(image); //Makes this component display an image.
    }

    ...//In a mouse-dragged event handler:
    Rectangle totalRepaint =
        rectToDraw.union(previousRectDrawn);
    repaint(totalRepaint.x, totalRepaint.y,
            totalRepaint.width,
            totalRepaint.height);

    ...
    public void paintComponent(Graphics g) {
        super.paintComponent(g); //paints the background+image
        //Paint a rectangle on top of the image.
        g.setColor(Color.white);
        g.drawRect(rectToDraw.x, rectToDraw.y,
                   rectToDraw.width - 1,
                   rectToDraw.height - 1);
    }
}
```

As you can see, the custom component extends JLabel so that it inherits the ability to display an image. The user can select a rectangular area by dragging the mouse. The component continuously displays a rectangle indicating the size of the current selection. To improve rendering speed, the component's mouse-dragged event handler specifies a painting area to the repaint method.

By limiting the area to be repainted, the event handlers avoid unnecessarily repainting the image outside of that area. For this small image there's no noticeable performance benefit to this strategy. However, for a large image there might be a real benefit. And if instead of painting an image from a file, you had to compute what to paint under the rectangle -- for example, computing shapes in a draw program -- then using knowledge of the paint area to limit the computation you perform might improve performance significantly.

The area specified to repaint must include not only the area to be painted, but also any area that needs to be erased. Otherwise, old painting remains visible until it happens to be erased by other painting. The preceding code calculates the total area to be repainted by taking the union of the rectangle to be painted with the rectangle that was previously painted.
The painting area specified to repaint is reflected in the Graphics object passed into the paintComponent method. You can use the getClipBounds method to determine which rectangular area to paint. Here is an example of using the clip bounds:

```java
public void paintComponent(Graphics g) {
    Rectangle clipRect = g.getClipBounds();
    if (clipRect != null) {
        // If it's more efficient, draw only the area
        // specified by clipRect.
        // Top-leftmost point = (clipRect.x, clipRect.y)
        // Width, height = clipRect.width, clipRect.height
        } else {
        // Paint the entire component.
    }
}
```

3.7 The Graphics Object

The Graphics object passed into the paintComponent method provides both a context for painting and methods for performing the painting. The methods, which we discuss in detail a little later, have names such as drawImage, drawString, drawRect, and fillRect.

The graphics context consists of state such as the current painting color, the current font, and (as you've already seen) the current painting area. The color and font are initialized to the foreground color and font of the component just before the invocation of paintComponent. You can get them using the getColor and getFont methods, and set them using the setColor and setFont methods.

You can safely ignore the current painting area, if you like. It has no effect on the component's coordinate system, and any painting outside the area is ignored. However, if your painting code involves complex operations that can be simplified if the painting area is reduced, then you should use your knowledge of the painting area to help you improve painting performance. As shown by the previous code example, you get the painting area's rectangular bounds from the Graphics object by invoking the getClipBounds method.

You can reduce the painting area in two ways. The first is to specify repaint with arguments whenever possible. The other is to implement paintComponent so that it invokes the Graphics object's setClip method. If you use setClip, be sure to restore the original painting area before returning. Otherwise, the component could be painted improperly. Here's an example of reducing and then restoring the painting area:

```java
Rectangle oldClipBounds = g.getClipBounds();
```
Rectangle clipBounds = new Rectangle(...);
g.setClip(clipBounds);

...//Perform custom painting...

g.setClip(oldClipBounds);

When writing your painting code, keep in mind that you can't depend on any graphics context except what's provided by the Graphics object. For example, you can't rely on the painting area you specify with repaint being exactly the same as the painting area used in the subsequent call to paintComponent. For one thing, multiple repaint requests can be coalesced into a single paintComponent call, with the painting area adjusted accordingly. For another, the painting system occasionally calls paintComponent on its own, without any repaint request from your program. As an example, the painting system invokes a component's paintComponent method when it first shows the component's GUI. Also, when the GUI is covered by another window and then becomes uncovered, the painting system invokes the paintComponent method with the painting area equal to the newly uncovered area.

3.8 The Swing Painting Methods

The paintComponent method is one of three methods that JComponent objects use to paint themselves. The three methods are invoked in this order:

1. paintComponent -- The main method for painting. By default, it first paints the background if the component is opaque. Then it performs any custom painting.
2. paintBorder -- Tells the component's border (if any) to paint. Do not invoke or override this method.
3. paintChildren -- Tells any components contained by this component to paint themselves. Do not invoke or override this method.

Note: Don't override or invoke the method that calls the paintXxx methods: the paint method. Although overriding paint was legitimate in pre-Swing components, it's generally not a good thing to do in components that descend from JComponent. Unless you're careful, overriding paint would likely confuse the painting system, which relies on the JComponent implementation of the paint method for correct painting, performance enhancements, and features such as double buffering.

The standard Swing components delegate their look-and-feel-specific painting to an object called a UI delegate. When such a component's paintComponent method is called, the method asks the UI delegate to paint the component. Generally, the UI delegate first checks whether the
component is opaque and, if so, paints the entire background of the component. Then the UI delegate performs any look-and-feel-specific painting.

The reason that we recommend extending JPanel instead of JComponent is that the JComponent class doesn't currently set up a UI delegate -- only its subclasses do. This means that if you extend JComponent, your component's background won't be painted unless you paint it yourself. When you extend JPanel and invoke super.paintComponent at the top of your paintComponent method, however, then the panel's UI delegate paints the component's background if the component is opaque.

3.9 Painting Shapes

The Graphics class defines methods for painting the following kinds of shapes:

- Lines (drawLine)
- Rectangles (drawRect and fillRect)
- Raised or lowered rectangles (draw3DRect and fill3DRect)
- Round-edged rectangles (drawRoundRect and fillRoundRect)
- Ovals (drawOval and fillOval)
- Arcs (drawArc and fillArc)
- Polygons (drawPolygon, drawPolyline, and fillPolygon)

Here is an example of painting the outline of a rectangle:

```
g.drawRect(x, y, rectWidth - 1, rectHeight - 1);
```

Here is an example of painting a filled rectangle of the same size.
```
g.fillRect(x, y, rectWidth, rectHeight);
```

Note that for the drawRect method, you must specify one pixel less than the desired width and height. This is because the painting system draws lines just below the specified rectangle, instead of within the specified rectangle. The same rule of specifying one less than the desired width applies to other drawXxx methods, such as draw3DRect. For the fillXxx methods, on the other hand, you specify exactly the desired width and height in pixels.

**Java™ 2 Note:** If you are using Java 2 (JDK™ 1.2), you can use the new Java™ 2D API, which allows you to create virtually any kind of geometric shape and to specify line styles, line sizes, and fancy fill patterns.
Example 1: Simple Rectangle Painting

Here's a picture of a program that's almost the same as the CoordinatesDemo program shown in The Coordinate System. Like CoordinatesDemo, this program paints a rectangle wherever the user clicks. However, this program's rectangle is larger and has a yellow fill. Here is a picture of its GUI:

![Click occurred at coordinate (100, 15).]

The program features two components. The largest is a custom component implemented by a class named RectangleArea. It paints the beveled border and everything inside it, including the yellow rectangle. The other component is a label that appears at the bottom of the GUI, under the custom component. The label describes the program's current state.

Here is the painting-related code for the custom component:

```java
class RectangleArea extends JPanel {
    ...
    int rectWidth = 50;
    int rectHeight = 50;
    ...
    public RectangleArea(...) {
        ...
        Border raisedBevel = BorderFactory.createRaisedBevelBorder();
        Border loweredBevel = BorderFactory.createLoweredBevelBorder();
        Border compound = BorderFactory.createCompoundBorder(raisedBevel,
                                                               loweredBevel);
        setBorder(compound);
        ...
    }
    ...

    public void paintComponent(Graphics g) {
        super.paintComponent(g); // paint background

        // Paint a filled rectangle at user's chosen point.
        if (point != null) {
```
g.drawRect(point.x, point.y, rectWidth - 1, rectHeight - 1);
g.setColor(Color.yellow);
g.fillRect(point.x + 1, point.y + 1, rectWidth - 2, rectHeight - 2);
controller.updateLabel(point);
}
}
}

The component's implementation of paintComponent uses the fillRect method to paint a 50-by-50-pixel rectangle outline, filled with a 48-by-48-pixel yellow rectangle. Note the differences in the arguments specified to drawRect and fillRect.

**Note:** It's perfectly legal to specify x, y, height, or width values that are negative or cause a result larger than the painting area. Values outside the painting area don't matter too much because they're clipped to the painting area. You just won't see part of the shape. Negative height or width results in the shape not being painted at all.

For a little more information about this example, see [The Coordinate System](#), which features the CoordinatesDemo example on which RectangleDemo is based.

### Example 2: A Shape Sampler

The ShapesDemo program demonstrates all the shapes you can draw and fill, using API supported with both JDK 1.1 and Java 2. Here is a picture of its GUI:

![ShapesDemo GUI](#)

**Note:** Unless the default font is very small, some of the strings displayed by ShapesDemo overlap with other strings. A fix for this problem is demonstrated in [Getting Information About a Font: FontMetrics](#).

You can find the code for the entire program in ShapesDemo.java. The following snippet is just the code that paints the geometric shapes, where the bold lines are the actual invocations of painting methods. The rectHeight and rectWidth variables specify the size in pixels of the rectangle that contains the shape to be drawn. The x and y variables are changed for every shape, so that the shapes aren't painted on top of each other. The bg and fg variables are Color.
objects that specify the component's background and foreground colors, respectively.

```java
Color fg3D = Color.lightGray;
...
// drawLine(x1, y1, x2, y2)
g.drawLine(x, y+rectHeight-1, x + rectWidth, y);
...
// drawRect(x, y, w, h)
g.drawRect(x, y, rectWidth, rectHeight);
...
// draw3DRect(x, y, w, h, raised)
g.setColor(fg3D);
g.draw3DRect(x, y, rectWidth, rectHeight, true);
g.setColor(fg);
...
// drawRoundRect(x, y, w, h, arcw, arch)
g.drawRoundRect(x, y, rectWidth, rectHeight, 10, 10);
...
// drawOval(x, y, w, h)
g.drawOval(x, y, rectWidth, rectHeight);
...
// drawArc(x, y, w, h, startAngle, arcAngle)
g.drawArc(x, y, rectWidth, rectHeight, 90, 135);
...
// drawPolygon(xPoints, yPoints, numPoints)
int x1Points[] = {x, x+rectWidth, x, x+rectWidth};
int y1Points[] = {y, y+rectHeight, y+rectHeight, y};
g.drawPolygon(x1Points, y1Points, x1Points.length);
...
// drawPolyline(xPoints, yPoints, numPoints)
// Note: drawPolygon would close the polygon.
int x2Points[] = {x, x+rectWidth, x, x+rectWidth};
int y2Points[] = {y, y+rectHeight, y+rectHeight, y};
g.drawPolyline(x2Points, y2Points, x2Points.length);
...
// fillRect(x, y, w, h)
g.fillRect(x, y, rectWidth, rectHeight);
...
// fill3DRect(x, y, w, h, raised)
g.setColor(fg3D);
g.fill3DRect(x, y, rectWidth, rectHeight, true);
g.setColor(fg);
...
// fillRoundRect(x, y, w, h, arcw, arch)
```
g.fillRoundRect(x, y, rectWidth, rectHeight, 10, 10);
...
// fillOval(x, y, w, h)
g.fillOval(x, y, rectWidth, rectHeight);
...
// fillArc(x, y, w, h, startAngle, arcAngle)
g.fillArc(x, y, rectWidth, rectHeight, 90, 135);
...
// fillPolygon(xPoints, yPoints, numPoints)
int x3Points[] = {x, x+rectWidth, x, x+rectWidth};
int y3Points[] = {y, y+rectHeight, y+rectHeight, y};
g.fillPolygon(x3Points, y3Points, x3Points.length);
...

4 How to Use Borders

Every JComponent can have one or more borders. Borders are incredibly useful objects that, while not themselves components, know how to draw the edges of Swing components. Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components. To put a border around a JComponent, you use its setBorder method. You can use the BorderFactory class to create most of the borders that Swing provides. Here is an example of code that creates a bordered container:

```java
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createLineBorder(Color.black));
```

Here's a picture of the container, which contains a label component. The black line drawn by the border marks the edge of the container.

![line border]

The rest of this page discusses the following topics:
- The BorderDemo Example
- Using the Borders Provided by Swing
- Creating Custom Borders
- Adding a Border to a Bordered Swing Component
- The Border API
- Examples of Using Borders
4.1 The BorderDemo example

The following pictures show an application called BorderDemo that displays the borders Swing provides. We show the code for creating these borders a little later, in Using the Borders Provided by Swing.

The next picture shows some matte borders. When creating a matte border, you specify how many pixels it occupies at the top, left, bottom, and right of a component. You then specify either a color or an icon for the matte border to draw. You need to be careful when choosing the icon and determining your component's size; otherwise, the icon might get chopped off or have mismatch at the component's corners.

The next picture shows titled borders. Using a titled border, you can convert any border into one that displays a text description. If you don't specify a border, then a look-and-feel-specific border is used. For example, the default titled border in
the Java look and feel uses a gray line, and the default titled border in the
Windows look and feel uses an etched border. By default, the title straddles the
upper left of the border, as shown at the top of the following figure.

The next picture shows compound borders. With compound borders, you can
combine any two borders, which can themselves be compound borders.

4.2 Using the Borders Provided by Swing

The code that follows shows how to create and set the borders you saw in the
preceding figures. You can find the program's code in BorderDemo.java. To run the program, you will also need to put an image file in a directory named
images: left.gif.

//Keep references to the next few borders, for use in titles
//and compound borders.
Border blackline, etched, raisedbevel, loweredbevel, empty;
blackline = BorderFactory.createLineBorder(Color.black);
etched = BorderFactory.createEtchedBorder();
raisedbevel = BorderFactory.createRaisedBevelBorder();
loweredbevel = BorderFactory.createLoweredBevelBorder();
empty = BorderFactory.createEmptyBorder();

//Simple borders
jComp2.setBorder(blackline);
jComp3.setBorder(raisedbevel);
jComp4.setBorder(loweredbevel);
jComp5.setBorder(empty);

//Matte borders
ImageIcon icon = new ImageIcon("images/left.gif"); //20x22
jComp6.setBorder(BorderFactory.createMatteBorder(
    -1, -1, -1, -1, icon));
jComp7.setBorder(BorderFactory.createMatteBorder(
    1, 5, 1, 1,
    Color.red));
jComp8.setBorder(BorderFactory.createMatteBorder(
    0, 20, 0, 0, icon));

//Titled borders
TitledBorder title1, title2, title3, title4, title5;
title1 = BorderFactory.createTitledBorder("title");
jComp9.setBorder(title1);

title2 = BorderFactory.createTitledBorder(
    blackline, "title");
title2.setTitleJustification(TitledBorder.CENTER);
jComp10.setBorder(title2);

title3 = BorderFactory.createTitledBorder(
    etched, "title");
title3.setTitleJustification(TitledBorder.RIGHT);
jComp11.setBorder(title3);

title4 = BorderFactory.createTitledBorder(
    loweredbevel, "title");
title4.setTitlePosition(TitledBorder.ABOVE_TOP);
jComp12.setBorder(title4);

title5 = BorderFactory.createTitledBorder(
    empty, "title");
title5.setTitlePosition(TitledBorder.BOTTOM);
jComp13.setBorder(title5);

//Compound borders
Border compound1, compound2, compound3;
Border redline = BorderFactory.createLineBorder(Color.red);
//This creates a nice frame.
compound1 = BorderFactory.createCompoundBorder(
    raisedbevel, loweredbevel);
jComp14.setBorder(compound1);

//Add a red outline to the frame.
compound2 = BorderFactory.createCompoundBorder(
    redline, compound1);
jComp15.setBorder(compound2);

//Add a title to the red-outlined frame.
compound3 = BorderFactory.createTitledBorder(
    compound2, "title",
    TitledBorder.CENTER,
    TitledBorder.BELOW_BOTTOM);
jComp16.setBorder(compound3);

As you probably noticed, the code uses the BorderFactory class to create
each border. The BorderFactory class, which is in the javax.swing
package, returns objects that implement the Border interface.
The Border interface, as well as its Swing-provided implementations, is in the
javax.swing.border package. You often don't need to directly use
anything in the border package, except when specifying constants that are specific
to a particular border class or when referring to the Border type.

4.3 Creating Custom Borders

If BorderFactory doesn't offer you enough control over a border's form, then
you might need to directly use the API in the border package -- or even define
your own border. In addition to containing the Border interface, the border
package contains the classes that implement the borders you've already seen:
LineBorder, EtchedBorder, BevelBorder, EmptyBorder, MatteBorder, TitledBorder, and CompoundBorder. The border
package also contains a class named SoftBevelBorder, which produces a
result similar to BevelBorder, but with softer edges.
If none of the Swing borders is suitable, you can implement your own border.
Generally, you do this by creating a subclass of the AbstractBorder class.
In your subclass, you must implement at least one constructor and the following
two methods:
• paintBorder, which contains the drawing code that a JComponent
  executes to draw the border.
• getBorderInsets, which specifies the amount of space the border
  needs to draw itself.
In addition, if your border is opaque, you might be able to decrease component
drawing time by overriding the border's isBorderOpaque method so that it
returns true. For examples of implementing borders, see the source code for the
classes in the javax.swing.border package.

4.4 Adding a Border to a Bordered Swing Component

Many of the ready-to-use Swing components use borders to draw the outline of
the component. If you want to draw an additional border around an already
bordered component -- to provide some extra space above a scroll pane, for
example -- then you need to add the new border to the existing border. Be sure to
test the component to make sure that it works well with your extra border;
components that change their border depending on their state probably aren't good
candidates for additional borders. Here's an example of adding to an existing
border:

```java
aJComponent.setBorder(
    BorderFactory.createCompoundBorder(
        BorderFactory.createEmptyBorder(20,0,0,0),
        aJComponent.getBorder())
);
```

The new border returned by createEmptyBorder adds 20 pixels of empty
space above any component that uses it. The code uses the
createCompoundBorder method to combine the new border with the
existing border, which is returned by getBorder.

4.5 The Border API

The following tables list the commonly used border methods. The API for using
borders falls into two categories:

- Creating a Border with BorderFactory
- Setting or Getting a Component's Border

<table>
<thead>
<tr>
<th>Creating a Border with BorderFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Method</strong></td>
</tr>
<tr>
<td>Border createLineBorder(Color)</td>
</tr>
<tr>
<td>Border createLineBorder(Color, int)</td>
</tr>
<tr>
<td>Border createEtchedBorder()</td>
</tr>
<tr>
<td>Border createEtchedBorder(Color, Color)</td>
</tr>
<tr>
<td>Border</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td><code>createLoweredBevelBorder()</code></td>
</tr>
<tr>
<td><code>createRaisedBevelBorder()</code></td>
</tr>
<tr>
<td><code>createBevelBorder(int, Color, Color)</code></td>
</tr>
<tr>
<td><code>createEmptyBorder()</code></td>
</tr>
<tr>
<td><code>createMatteBorder(int, int, int, int, Color)</code></td>
</tr>
<tr>
<td><code>createTitledBorder(String)</code></td>
</tr>
<tr>
<td><code>createCompoundBorder(Border, Border)</code></td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------</td>
</tr>
</tbody>
</table>
| void setBorder(Border)  
Border getBorder() | Set or get the border of the receiving JComponent. |
| void setBorderPainted(boolean)  
boolean isBorderPainted()  
(in AbstractButton, JMenuBar, JPopupMenu, JProgressBar, and JToolBar) | Set or get whether the border of the component should be painted. |