Chapter 7

# Graphics Programming



- Introducing Swing
- Creating a Frame
- Positioning a Frame
- Displaying Information in a Component
- Working with 2D Shapes
- Using Color
- Using Special Fonts for Text
- Displaying Images

To this point, you have seen only how to write programs that take input from the keyboard, fuss with it, and then display the results on a console screen. This is not what most users want now. Modern programs don't work this way and neither do web pages. This chapter starts you on the road to writing Java programs that use a graphical user interface (GUI). In particular, you learn how to write programs that size and locate windows on the screen, display text with multiple fonts in a window, display images, and so on. This gives you a useful, valuable repertoire of skills that you will put to good use in subsequent chapters as you write interesting programs.

The next two chapters show you how to process events, such as keystrokes and mouse clicks, and how to add interface elements, such as menus and buttons, to your applications. When you finish these three chapters, you will know the essentials for writing graphical applications. For more sophisticated graphics programming techniques, we refer you to Volume II.

If, on the other hand, you intend to use Java for server-side programming only and are not interested in writing GUI programming, you can safely skip these chapters.

## Introducing Swing

When Java 1.0 was introduced, it contained a class library, which Sun called the Abstract Window Toolkit (AWT), for basic GUI programming. The basic AWT library deals with user interface elements by delegating their creation and behavior to the native GUI toolkit on each target platform (Windows, Solaris, Macintosh, and so on). For example, if you used the original AWT to put a text box on a Java window, an underlying "peer" text box actually handled the text input. The resulting program could then, in theory, run on any of these platforms, with the "look and feel" of the target platform—hence Sun's trademarked slogan "Write Once, Run Anywhere."

The peer-based approach worked well for simple applications, but it soon became apparent that it was fiendishly difficult to write a high-quality portable graphics library that depended on native user interface elements. User interface elements such as menus, scrollbars, and text fields can have subtle differences in behavior on different platforms. It was hard, therefore, to give users a consistent and predictable experience with this approach. Moreover, some graphical environments (such as X11/Motif) do not have as rich a collection of user interface components as does Windows or the Macintosh. This in turn further limits a portable library based on peers to a "lowest common denominator" approach. As a result, GUI applications built with the AWT simply did not look as nice as native Windows or Macintosh applications, nor did they have the kind of functionality that users of those platforms had come to expect. More depressingly, there were *different* bugs in the AWT user interface library on the different platforms. Developers complained that they needed to test their applications on each platform, a practice derisively called "write once, debug everywhere."

In 1996, Netscape created a GUI library they called the IFC (Internet Foundation Classes) that used an entirely different approach. User interface elements, such as buttons, menus, and so on, were *painted* onto blank windows. The only functionality required from the underlying windowing system was a way to put up windows and to paint on the window. Thus, Netscape's IFC widgets looked and behaved the same no matter which platform the program ran on. Sun worked with Netscape to perfect this approach, creating a user interface library with the code name "Swing." Swing was available as an extension to Java 1.1 and became a part of the standard library in Java SE 1.2.

Since, as Duke Ellington said, "It Don't Mean a Thing If It Ain't Got That Swing," Swing is now the official name for the non-peer-based GUI toolkit. Swing is part of the Java Foundation Classes (JFC). The full JFC is vast and contains far more than the Swing GUI toolkit. JFC features not only include the Swing components but also an accessibility API, a 2D API, and a drag-and-drop API.

NOTE: Swing is not a complete replacement for the AWT—it is built on top of the AWT architecture. Swing simply gives you more capable user interface components. You use the foundations of the AWT, in particular, event handling, whenever you write a Swing program. From now on, we say "Swing" when we mean the "painted" user interface classes, and we say "AWT" when we mean the underlying mechanisms of the windowing toolkit, such as event handling.

Of course, Swing-based user interface elements will be somewhat slower to appear on the user's screen than the peer-based components used by the AWT. Our experience is that on any reasonably modern machine, the speed difference shouldn't be a problem. On the other hand, the reasons to choose Swing are overwhelming:

- Swing has a rich and convenient set of user interface elements.
- Swing has few dependencies on the underlying platform; it is therefore less prone to platform-specific bugs.
- Swing gives a consistent user experience across platforms.

Still, the third plus is also a potential drawback: If the user interface elements look the same on all platforms, then they will look *different* from the native controls and thus users will be less familiar with them.

Swing solves this problem in a very elegant way. Programmers writing Swing programs can give the program a specific "look and feel." For example, Figures 7–1 and 7–2 show the same program running with the Windows and the GTK look and feel.

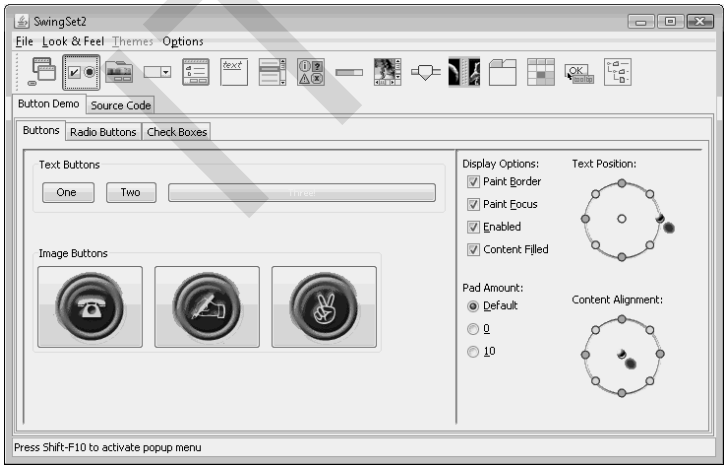


**Figure 7–1 The Windows look and feel of Swing**

Furthermore, Sun developed a platform-independent look and feel that was called "Metal" until the marketing folks renamed it as the "Java look and feel." However, most programmers continue to use the term "Metal," and we will do the same in this book.

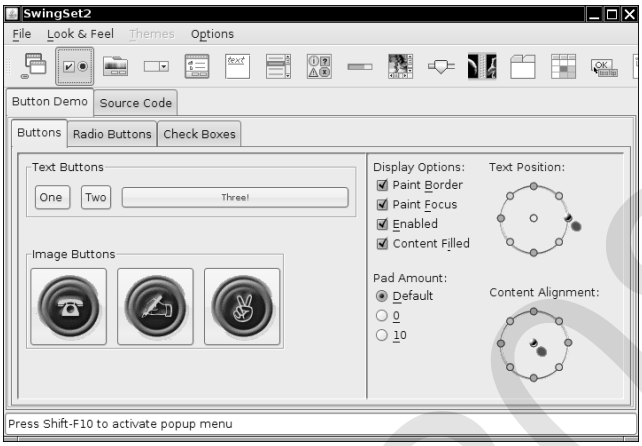


**Figure 7–2 The GTK look and feel of Swing**

Some people criticized Metal as being stodgy, and the look was freshened up for the Java SE 5.0 release (see Figure 7–3). Now the Metal look supports multiple themes—minor variations in colors and fonts. The default theme is called "Ocean."

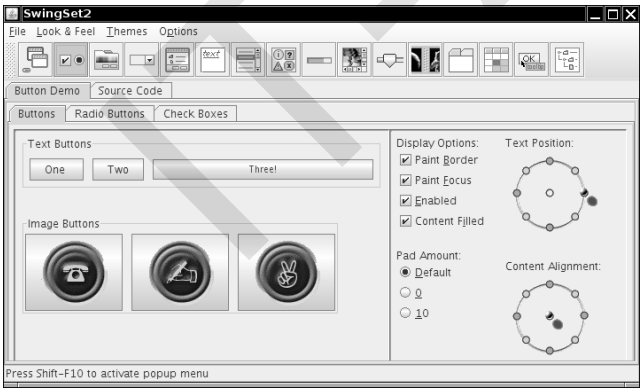


**Figure 7–3 The Ocean theme of the Metal look and feel**

In Java SE 6, Sun improved the support for the native look and feel for Windows and GTK. A Swing application will now pick up color scheme customizations and faithfully render the throbbing buttons and scrollbars that have become fashionable.

Some users prefer that their Java applications use the native look and feel of their platforms, others like Metal or a third-party look and feel. As you will see in Chapter 8, it is very easy to let your users choose their favorite look and feel

NOTE: Although we won't have space in this book to tell you how to do it, Java programmers can extend an existing look and feel or even design a totally new look and feel. This is a tedious process that involves specifying how each Swing component is painted. Some developers have done just that, especially when porting Java to nontraditional platforms such as kiosk terminals or handheld devices. See `http://www.javootoo.com` for a collection of interesting look-and-feel implementations.

Java SE 5.0 introduced a look and feel, called Synth, that makes this process easier. In Synth, you can define a new look and feel by providing image files and XML descriptors, without doing any programming.

NOTE: Most Java user interface programming is nowadays done in Swing, with one notable exception. The Eclipse integrated development environment uses a graphics toolkit called SWT that is similar to the AWT, mapping to native components on various platforms. You can find articles describing SWT at `http://www.eclipse.org/articles/`.

If you have programmed Microsoft Windows applications with Visual Basic or C#, you know about the ease of use that comes with the graphical layout tools and resource editors these products provide. These tools let you design the visual appearance of your application, and then they generate much (often all) of the GUI code for you. GUI builders are available for Java programming, but we feel that in order to use these tools effectively, you should know how to build a user interface manually. The remainder of this chapter tells you the basics about displaying a window and painting its contents.

## Creating a Frame

A top-level window (that is, a window that is not contained inside another window) is called a *frame* in Java. The AWT library has a class, called `Frame`, for this top level. The Swing version of this class is called `JFrame` and extends the `Frame` class. The `JFrame` is one of the few Swing components that is not painted on a canvas. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user's windowing system, not by Swing.

CAUTION: Most Swing component classes start with a "J": `JButton`, `JFrame`, and so on. There are classes such as `Button` and `Frame`, but they are AWT components. If you accidentally omit a "J", your program may still compile and run, but the mixture of Swing and AWT components can lead to visual and behavioral inconsistencies.

In this section, we go over the most common methods for working with a Swing *JFrame*. Listing 7–1 lists a simple program that displays an empty frame on the screen, as illustrated in Figure 7–4.

**Figure 7–4 The simplest visible frame**

**Listing 7–1** SimpleFrameTest.java

```
1. import javax.swing.*;
2.
3. /**
4.  * @version 1.32 2007-06-12
5.  * @author Cay Horstmann
6.  */
7. public class SimpleFrameTest
8. {
9.    public static void main(String[] args)
10.   {
11.      SimpleFrame frame = new SimpleFrame();
12.      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13.      frame.setVisible(true);
14.   }
15. }
16.
17. class SimpleFrame extends JFrame
18. {
19.    public SimpleFrame()
20.    {
21.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22.    }
23.
24.    public static final int DEFAULT_WIDTH = 300;
25.    public static final int DEFAULT_HEIGHT = 200;
26. }
```

Let's work through this program, line by line.

The Swing classes are placed in the `javax.swing` package. The package name `javax` indicates a Java extension package, not a core package. For historical reasons, Swing is considered an extension. However, it is present in every Java SE implementation since version 1.2.

By default, a frame has a rather useless size of 0 × 0 pixels. We define a subclass `SimpleFrame` whose constructor sets the size to 300 × 200 pixels. This is the only difference between a `SimpleFrame` and a `JFrame`.

In the `main` method of the `SimpleFrameTest` class, we construct a `SimpleFrame` object and make it visible.

There are two technical issues that we need to address in every Swing program.

First, all Swing components must be configured from the *event dispatch thread*, the thread of control that passes events such as mouse clicks and keystrokes to the user interface components. The following code fragment is used to execute statements in the event dispatch thread:

```
EventQueue.invokeLater(new Runnable()
   {
      public void run()
      {
         statements
      }
   });
```

We discuss the details in Chapter 14. For now, you should simply consider it a magic incantation that is used to start a Swing program.

NOTE: You will see many Swing programs that do not initialize the user interface in the event dispatch thread. It used to be perfectly acceptable to carry out the initialization in the main thread. Sadly, as Swing components got more complex, the programmers at Sun were no longer able to guarantee the safety of that approach. The probability of an error is extremely low, but you would not want to be one of the unlucky few who encounter an intermittent problem. It is better to do the right thing, even if the code looks rather mysterious.

Next, we define what should happen when the user closes the application's frame. For this particular program, we want the program to exit. To select this behavior, we use the statement

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

In other programs with multiple frames, you would not want the program to exit just because the user closes one of the frames. By default, a frame is hidden when the user closes it, but the program does not terminate. (It might have been nice if the program terminated after the *last* frame became invisible, but that is not how Swing works.)

Simply constructing a frame does not automatically display it. Frames start their life invisible. That gives the programmer the chance to add components into the frame before showing it for the first time. To show the frame, the `main` method calls the `setVisible` method of the frame.

NOTE: Before Java SE 5.0, it was possible to use the `show` method that the `JFrame` class inherits from the superclass `Window`. The `Window` class has a superclass `Component` that also has a `show` method. The `Component.show` method was deprecated in Java SE 1.2. You are supposed to call `setVisible(true)` instead if you want to show a component. However, until Java SE 1.4, the `Window.show` method was *not* deprecated. In fact, it was quite useful, making the window visible *and* bringing it to the front. Sadly, that benefit was lost on the deprecation police, and Java SE 5.0 deprecated the `show` method for windows as well.

After scheduling the initialization statements, the `main` method exits. Note that exiting `main` does not terminate the program, just the main thread. The event dispatch thread keeps the program alive until it is terminated, either by closing the frame or by calling the `System.exit` method.

The running program is shown in Figure 7–4 on page 286—it is a truly boring top-level window. As you can see in the figure, the title bar and surrounding decorations, such as resize corners, are drawn by the operating system and not the Swing library. If you run the same program in Windows, GTK, or the Mac, the frame decorations are different. The Swing library draws everything inside the frame. In this program, it just fills the frame with a default background color.

NOTE: As of Java SE 1.4, you can turn off all frame decorations by calling `frame.setUndecorated(true)`.

## Positioning a Frame

The `JFrame` class itself has only a few methods for changing how frames look. Of course, through the magic of inheritance, most of the methods for working with the size and position of a frame come from the various superclasses of `JFrame`. Here are some of the most important methods:

- The `setLocation` and `setBounds` methods for setting the position of the frame
- The `setIconImage` method, which tells the windowing system which icon to display in the title bar, task switcher window, and so on
- The `setTitle` method for changing the text in the title bar
- The `setResizable` method, which takes a `boolean` to determine if a frame will be resizable by the user

Figure 7–5 illustrates the inheritance hierarchy for the `JFrame` class.

TIP: The API notes for this section give what we think are the most important methods for giving frames the proper look and feel. Some of these methods are defined in the `JFrame` class. Others come from the various superclasses of `JFrame`. At some point, you may need to search the API docs to see if there are methods for some special purpose. Unfortunately, that is a bit tedious to do with inherited methods. For example, the `toFront` method is applicable to objects of type `JFrame`, but because it is simply inherited from the `Window` class, the `JFrame` documentation doesn't explain it. If you feel that there should be a method to do something and it isn't explained in the documentation for the class you are working with, try looking at the API documentation for the methods of the *superclasses* of that class. The top of each API page has hyperlinks to the superclasses, and inherited methods are listed below the method summary for the new and overridden methods.

As the API notes indicate, the `Component` class (which is the ancestor of all GUI objects) and the `Window` class (which is the superclass of the `Frame` class) are where you need to look to find the methods to resize and reshape frames. For example, the `setLocation` method in the `Component` class is one way to reposition a component. If you make the call

```
setLocation(x, y)
```

the top-left corner is located `x` pixels across and `y` pixels down, where (0, 0) is the top-left corner of the screen. Similarly, the `setBounds` method in `Component` lets you resize and relocate a component (in particular, a `JFrame`) in one step, as

```
setBounds(x, y, width, height)
```

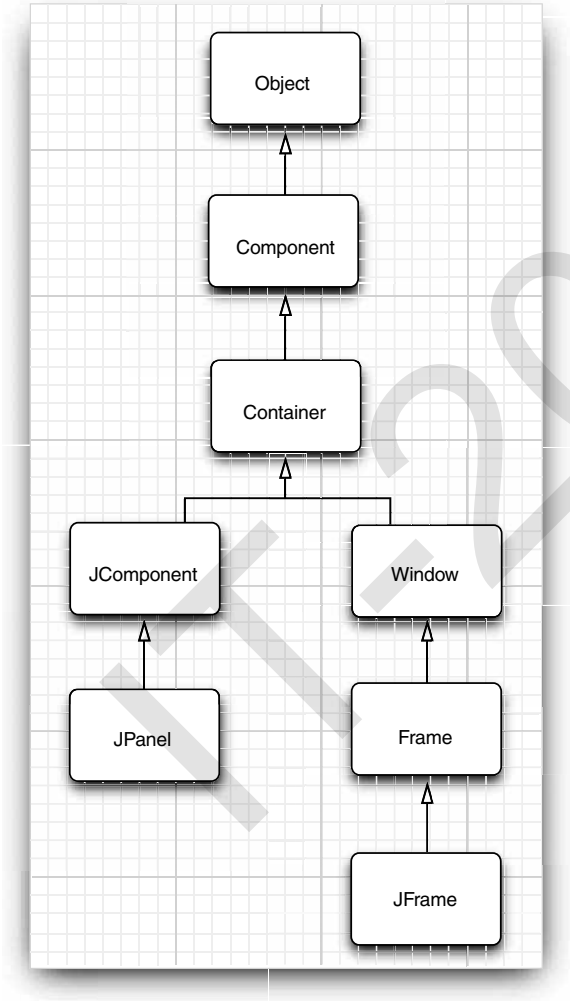


**Figure 7–5 Inheritance hierarchy for the frame and component classes in AWT and Swing**

Alternatively, you can give the windowing system control on window placement. If you call

```
setLoationByPlatform(true);
```

before displaying the window, the windowing system picks the location (but not the size), typically with a slight offset from the last window.

NOTE: For a frame, the coordinates of the `setLocation` and `setBounds` are taken relative to the whole screen. As you will see in Chapter 9, for other components inside a container, the measurements are taken relative to the container.

### Frame Properties

Many methods of component classes come in getter/setter pairs, such as the following methods of the `Frame` class:

```
public String getTitle()
public void setTitle(String title)
```

Such a getter/setter pair is called a *property*. A property has a name and a type. The name is obtained by changing the first letter after the `get` or `set` to lowercase. For example, the `Frame` class has a property with name `title` and type `String`.

Conceptually, `title` is a property of the frame. When we set the property, we expect that the title changes on the user's screen. When we get the property, we expect that we get back the value that we set.

We do not know (or care) how the `Frame` class implements this property. Perhaps it simply uses its peer frame to store the title. Perhaps it has an instance field

```
private String title; // not required for property
```

If the class does have a matching instance field, we don't know (or care) how the getter and setter methods are implemented. Perhaps they just read and write the instance field. Perhaps they do more, such as notifying the windowing system whenever the title changes.

There is one exception to the `get`/`set` convention: For properties of type `boolean`, the getter starts with `is`. For example, the following two methods define the `locationByPlatform` property:

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```

We will look at properties in much greater detail in Chapter 8 of Volume II.

NOTE: Many programming languages, in particular, Visual Basic and C#, have built-in support for properties. It is possible that a future version of Java will also have a language construct for properties.

### Determining a Good Frame Size

Remember: if you don't explicitly size a frame, all frames will default to being 0 by 0 pixels. To keep our example programs simple, we resize the frames to a size that we hope works acceptably on most displays. However, in a professional application,

## EXAMPLE: (A SIMPLE FRAME CREATION)

## METHOD:1(BY INHERITANCE)

```
package vijay.java.swing;

import java.awt.Frame;


public class MyFrame extends Frame{
	MyFrame(String s){
		super(s);
		setSize(400, 400);
		setVisible(true);
	}
	public static void main(String args[]){
		new MyFrame("Vijay verma");
	}

}
```

## METHOD:2(BY ASSOCIATION)

```
package vijay.java.swing;

import java.awt.Frame;


public class MyFrame {

	Frame f;
	MyFrame(String s){
		f=new Frame(s);
		f.setSize(400, 400);
		f.setVisible(true);

	}

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		new MyFrame("vijay");

	}

}
```

## EXAMPLE: (A FRAME WITH A TEXTFIELD & BUTTON)

```
package com.verma.vijay;

import java.awt.Button;
import java.awt.Frame;
import java.awt.TextField;

public class MyFrame2 {

	/**
	 * @param args
	 */
	Frame f;
	TextField tf;
	Button b1;
	MyFrame2(String s){
		f= new Frame(s);
		tf= new TextField();
		tf.setBounds(40, 40, 60, 30);
		f.add(tf);
		b1= new Button("ok");
		b1.setBounds(40, 400, 20, 20);
		f.add(b1);
		f.setLayout(null);
		f.setSize(400, 400);
		f.setVisible(true);

	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		new MyFrame2("verma");

	}

}
```