Pearson

Java核心技术

卷I：基础知识 下

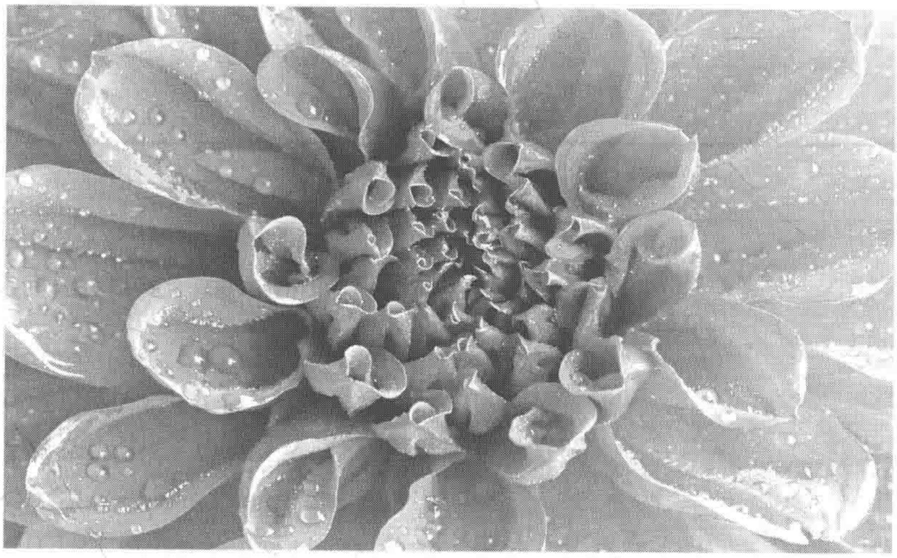
（第10版 · 英文版）

[美] Cay S. Horstmann 著

Core Java

Volume I—Fundamentals, Tenth Edition

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS



Java核心技术

卷I：基础知识 下

（第10版 · 英文版）

[美] Cay S. Horstmann 著

Core Java

Volume I—Fundamentals, Tenth Edition

人 民 邮 电 出 版 社

北 京

目录

CHAPTER 10

Graphics Programming

In this chapter

To this point, you have seen only how to write programs that take input from the keyboard, fuss with it, and 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 will 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 of 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.

10.1 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 depending 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 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 had 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; besides the Swing components, it also has 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. Whenever you write a Swing program, you use the foundations of the AWT—in particular, event handling. 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. In our experience, 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, they look *different* from the native controls, so 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 10.1 and 10.2 show the same program running with the Windows and the GTK look-and-feel.

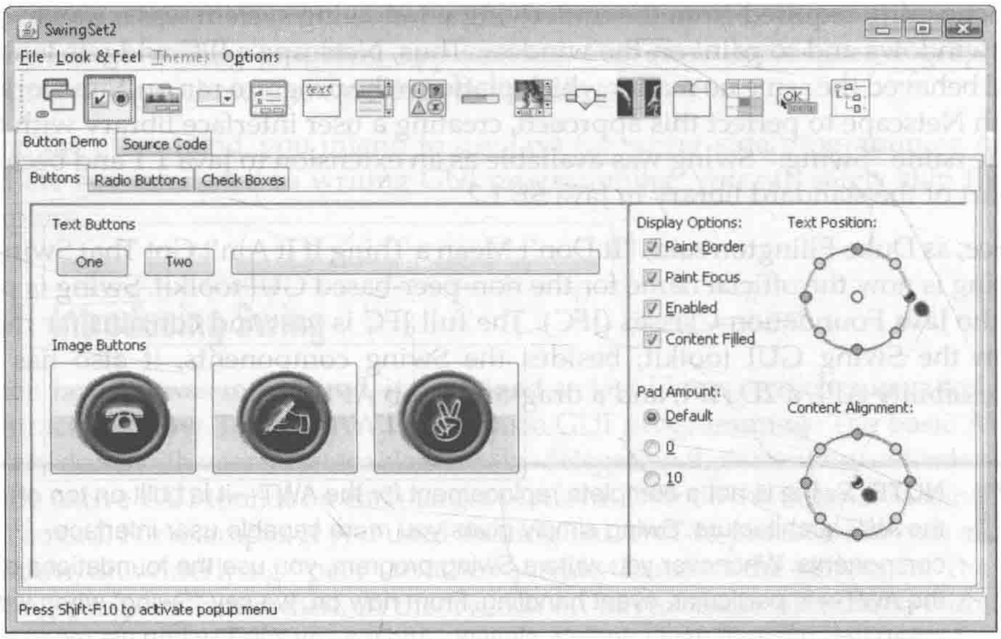


Figure 10.1 The Windows look-and-feel of Swing

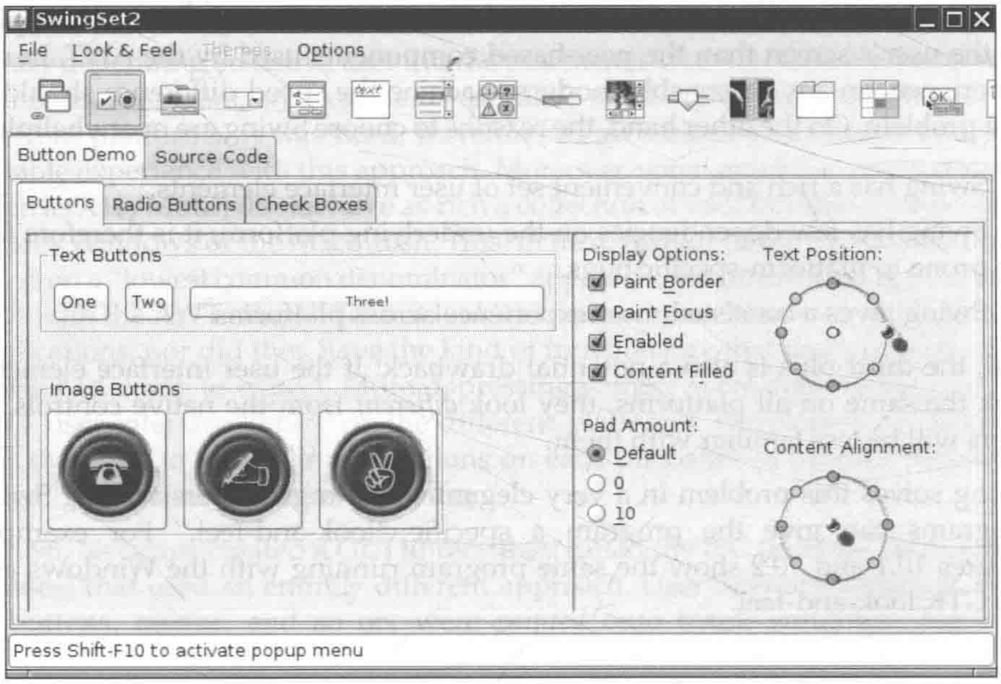


Figure 10.2 The GTK look-and-feel of Swing

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

Some people criticized Metal for being stodgy, and the look was freshened up for the Java SE 5.0 release (see Figure 10.3). Now the Metal look supports multiple themes—minor variations in colors and fonts. The default theme is called "Ocean."

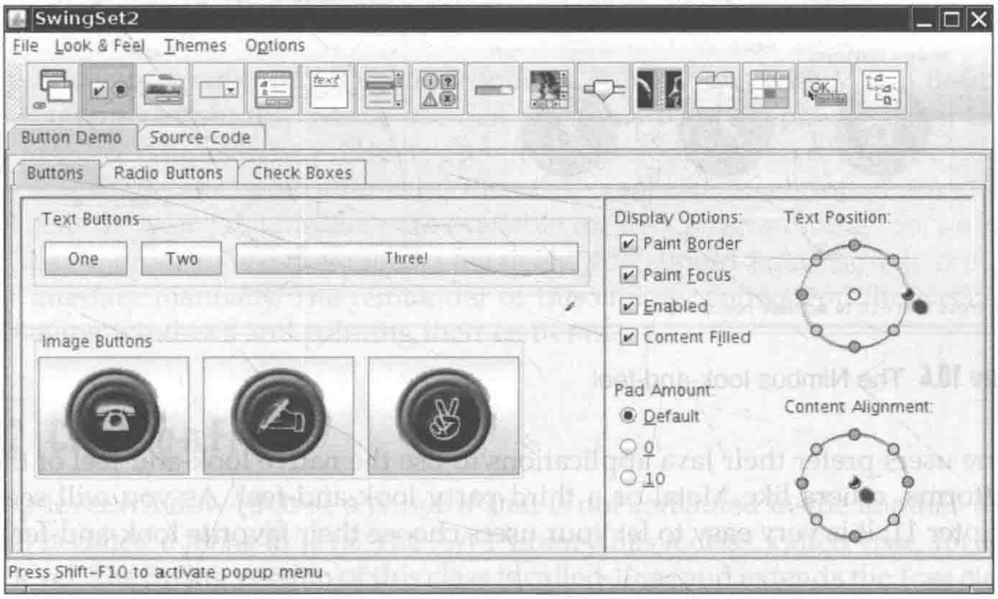
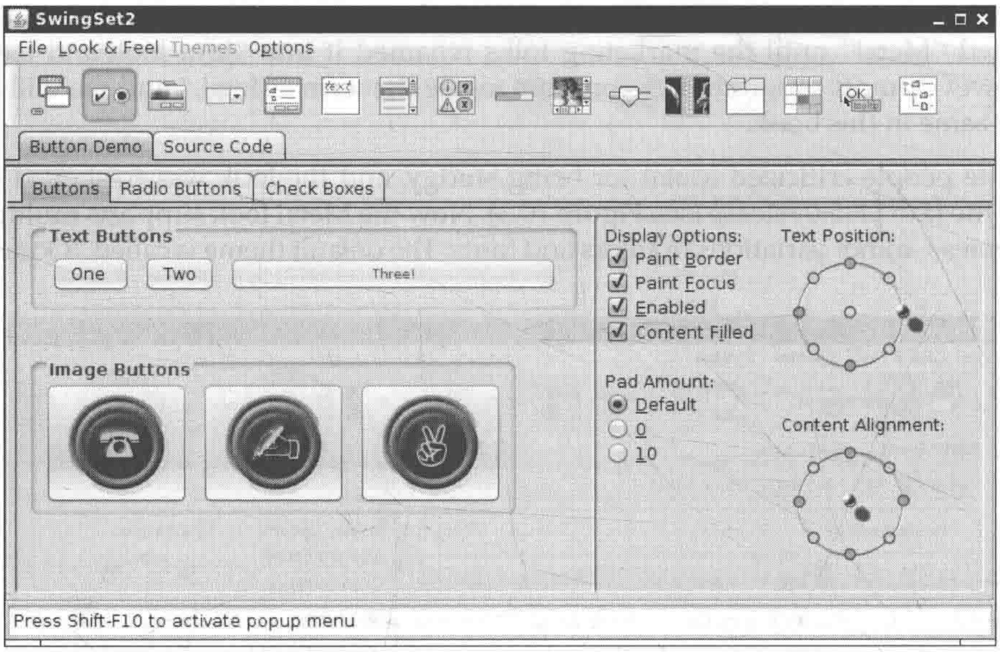


Figure 10.3 The Ocean theme of the Metal look-and-feel

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 the color scheme customizations and faithfully render the throbbing buttons and scrollbars that have become fashionable.

A new look-and-feel, called Nimbus (Figure 10.4), is offered since Java SE 7, but it is not available by default. Nimbus uses vector drawings, not bitmaps, and is therefore independent of the screen resolution.



Display Options:
- ✓ Paint Border
- ✓ Paint Focus
- ✓ Enabled
- ✓ Content Filled

Pad Amount:
- ● Default
- ○ 0
- ○ 10

Press Shift-F10 to activate popup menu

Figure 10.4 The Nimbus look-and-feel

Some users prefer their Java applications to 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 11, 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 one. 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 `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.

TIP: The Napkin look-and-feel (`http://napkinlaf.sourceforge.net`) gives a hand-drawn appearance to all user interface elements. This is very useful when you show prototypes to your customers, sending a clear message that you're not giving them a finished product.

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 the native components on various platforms. You can find articles describing SWT at `www.eclipse.org/articles`.

Oracle is developing an alternate technology, called JavaFX, as a replacement for Swing. We do not discuss JavaFX in this book. See `http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm` for more information.

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 too, 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 shows you the basics of displaying windows and painting their contents.

10.2 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 will go over the most common methods for working with a Swing `JFrame`. Listing 10.1 lists a simple program that displays an empty frame on the screen, as illustrated in Figure 10.5.

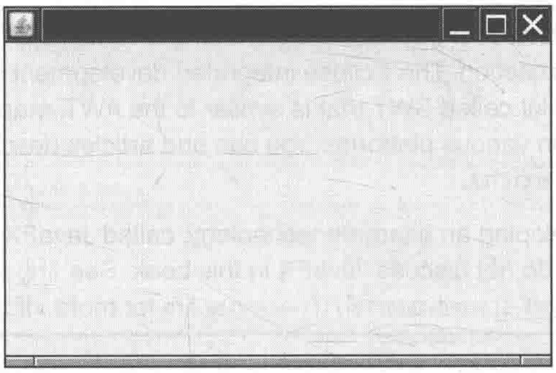


Figure 10.5 The simplest visible frame

Listing 10.1 simpleframe/SimpleFrameTest.java

```java
package simpleFrame;

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

/**
 * @version 1.33 2015-05-12
 * @author Cay Horstmann
 */
public class SimpleFrameTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() ->
         {
            SimpleFrame frame = new SimpleFrame();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         });
   }
}

class SimpleFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public SimpleFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

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(() ->
   {
      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 developers of the JDK 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