



Pearson



Java核心技术

卷I：基础知识

(第10版·英文版)

[美] Cay S. Horstmann 著

Core Java

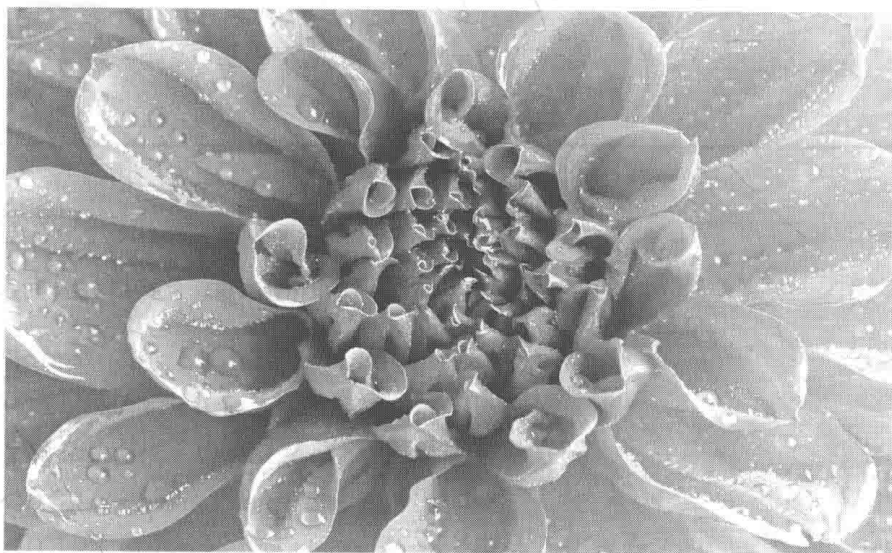
Volume I—Fundamentals, Tenth Edition



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

(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 / 图形界面编程	537
10.1 Introducing Swing / Swing 简介	538
10.2 Creating a Frame / 创建框架	543
10.3 Positioning a Frame / 设定框架的显示位置	546
10.3.1 Frame Properties / 框架属性	549
10.3.2 Determining a Good Frame Size / 确定合适的框架大小	549
10.4 Displaying Information in a Component / 在组件中显示信息	554
10.5 Working with 2D Shapes / 处理 2D 图形	560
10.6 Using Color / 使用颜色	569
10.7 Using Special Fonts for Text / 使用特殊的文本字体	573
10.8 Displaying Images / 显示图片	582
Chapter 11: Event Handling / 事件处理	587
11.1 Basics of Event Handling / 事件处理基础	587
11.1.1 Example: Handling a Button Click / 示例: 处理按钮点击事件	591
11.1.2 Specifying Listeners Concisely / 设置监听器的简洁方法	595
11.1.3 Example: Changing the Look-and-Feel / 示例: 修改观感	598
11.1.4 Adapter Classes / 适配器类	603
11.2 Actions / 动作	607
11.3 Mouse Events / 鼠标事件	616
11.4 The AWT Event Hierarchy / AWT 事件层次	624
11.4.1 Semantic and Low-Level Events / 语义与底层事件	626
Chapter 12: User Interface Components with Swing / Swing 用户界面组件	629
12.1 Swing and the Model-View-Controller Design Pattern / Swing 与模型-视图-控制器设计模式	630
12.1.1 Design Patterns / 设计模式	630

12.1.2	The Model-View-Controller Pattern / 模型-视图-控制器模式	632
12.1.3	A Model-View-Controller Analysis of Swing Buttons / Swing 按钮的模型-视图-控制器分析	636
12.2	Introduction to Layout Management / 布局管理简介	638
12.2.1	Border Layout / 边框布局	641
12.2.2	Grid Layout / 网格布局	644
12.3	Text Input / 文本输入	648
12.3.1	Text Fields / 文本框	649
12.3.2	Labels and Labeling Components / 标签与标签组件	651
12.3.3	Password Fields / 密码框	652
12.3.4	Text Areas / 文本区域	653
12.3.5	Scroll Panes / 滚动窗格	654
12.4	Choice Components / 选择组件	657
12.4.1	Checkboxes / 复选框	657
12.4.2	Radio Buttons / 单选按钮	660
12.4.3	Borders / 边框	664
12.4.4	Combo Boxes / 组合框	668
12.4.5	Sliders / 滑动条	672
12.5	Menus / 菜单	678
12.5.1	Menu Building / 菜单构建	679
12.5.2	Icons in Menu Items / 菜单项中的图标	682
12.5.3	Checkbox and Radio Button Menu Items / 复选框和单选按钮菜单项	683
12.5.4	Pop-Up Menus / 弹出菜单	684
12.5.5	Keyboard Mnemonics and Accelerators / 键盘助记符与快捷键	686
12.5.6	Enabling and Disabling Menu Items / 启用和禁用菜单项	689
12.5.7	Toolbars / 工具栏	694
12.5.8	Tooltips / 工具提示	696
12.6	Sophisticated Layout Management / 复杂的布局管理	699

12.6.1	The Grid Bag Layout / 网格布局管理	701
12.6.2	Group Layout / 组布局	713
12.6.3	Using No Layout Manager / 不使用布局管理器	723
12.6.4	Custom Layout Managers / 定制布局管理器	724
12.6.5	Traversal Order / 遍历顺序	729
12.7	Dialog Boxes / 对话框	730
12.7.1	Option Dialogs / 选项对话框	731
12.7.2	Creating Dialogs / 创建对话框	741
12.7.3	Data Exchange / 数据交换	746
12.7.4	File Dialogs / 文件对话框	752
12.7.5	Color Choosers / 颜色选择器	764
12.8	Troubleshooting GUI Programs / GUI 程序的问题定位	770
12.8.1	Debugging Tips / 调试技巧	770
12.8.2	Letting the AWT Robot Do the Work / 把工作交给 AWT Robot	774
Chapter 13: Deploying Java Applications / 部署 Java 应用程序		779
13.1	JAR Files / JAR 文件	780
13.1.1	Creating JARfiles / 创建 JAR 文件	780
13.1.2	The Manifest / 清单文件	781
13.1.3	Executable JAR Files / 可执行的 JAR 文件	782
13.1.4	Resources / 资源	783
13.1.5	Sealing / 封闭	787
13.2	Storage of Application Preferences / 应用偏好信息的存储	788
13.2.1	Property Maps / 属性映射	788
13.2.2	The Preferences API / Properties API	794
13.3	Service Loaders / 服务加载器	800
13.4	Applets / Applet	802
13.4.1	A Simple Applet / 一个简单的 Applet	803
13.4.2	The applet HTML Tag and Its Attributes / applet HTML 标记及其属性	808

13.4.3	Use of Parameters to Pass Information to Applets / 使用参数向 Applet 传递信息	810
13.4.4	Accessing Image and Audio Files / 访问图片和音频文件	816
13.4.5	The Applet Context / Applet 上下文	818
13.4.6	Inter-Applet Communication / Applet 间的通信	818
13.4.7	Displaying Items in the Browser / 在浏览器中显示信息	819
13.4.8	The Sandbox / 沙箱	820
13.4.9	Signed Code / 签名代码	822
13.5	Java Web Start / Java Web Start	824
13.5.1	Delivering a Java Web Start Application / 交付一个 Java Web Start 应用	824
13.5.2	The JNLP API / JNLP API	829
Chapter 14: Concurrency /并发		839
14.1	What Are Threads? / 什么是线程	840
14.1.1	Using Threads to Give Other Tasks a Chance / 使用线程为其他任务提供执行机会	846
14.2	Interrupting Threads / 中断线程	851
14.3	Thread States / 线程状态	855
14.3.1	New Threads / 新创建线程	855
14.3.2	Runnable Threads / 可运行线程	855
14.3.3	Blocked and Waiting Threads / 被阻塞线程与等待线程	856
14.3.4	Terminated Threads / 被终止的线程	857
14.4	Thread Properties / 线程属性	858
14.4.1	Thread Priorities / 线程优先级	858
14.4.2	Daemon Threads / 守护线程	859
14.4.3	Handlers for Uncaught Exceptions / 未捕获异常的处理程序	860
14.5	Synchronization / 同步	862
14.5.1	An Example of a Race Condition / 竞争条件的一个案例	862
14.5.2	The Race Condition Explained / 竞争条件详解	866
14.5.3	Lock Objects / 锁对象	868
14.5.4	Condition Objects / 条件对象	872
14.5.5	The synchronized Keyword / synchronized 关键字	878

14.5.6	Synchronized Blocks / 同步块	882
14.5.7	The Monitor Concept / 监视器概念	884
14.5.8	Volatile Fields / volatile 字段	885
14.5.9	Final Variables / final 变量	886
14.5.10	Atomics / 原子	886
14.5.11	Deadlocks / 死锁	889
14.5.12	Thread-Local Variables / 线程局部变量	892
14.5.13	Lock Testing and Timeouts / 锁测试与超时	893
14.5.14	Read / Write Locks / 读/写锁	895
14.5.15	Why the stop and suspend Methods Are Deprecated / 为什么弃用 stop 和 suspend 方法	896
14.6	Blocking Queues / 阻塞队列	898
14.7	Thread-Safe Collections / 线程安全的集合	905
14.7.1	Efficient Maps, Sets, and Queues / 高效的映射、集和队列	905
14.7.2	Atomic Update of Map Entries / 映射表项的原子更新	907
14.7.3	Bulk Operations on Concurrent Hash Maps / 并发散列映射上的主要操作	909
14.7.4	Concurrent Set Views / 并发的集视图	912
14.7.5	Copy on Write Arrays / 写时复制的数组	912
14.7.6	Parallel Array Algorithms / 并行数组算法	912
14.7.7	Older Thread-Safe Collections / 较早的线程安全的集合	914
14.8	Callables and Futures / Callable 与 Future	915
14.9	Executors / 执行器	920
14.9.1	Thread Pools / 线程池	921
14.9.2	Scheduled Execution / 预订执行	926
14.9.3	Controlling Groups of Tasks / 控制任务组	927
14.9.4	The Fork-Join Framework / Fork-Join 框架	928
14.9.5	Completable Futures / CompletableFuture	931
14.10	Synchronizers / 同步器	934
14.10.1	Semaphores / 信号量	935
14.10.2	Countdown Latches / 倒计时门栓	936
14.10.3	Barriers / 障栅	936
14.10.4	Exchangers / 交换器	937

14.10.5	Synchronous Queues / 同步队列	937
14.11	Threads and Swing / 线程与 Swing	937
14.11.1	Running Time-Consuming Tasks / 运行耗时任务	939
14.11.2	Using the Swing Worker / 使用 Swing 工作线程	943
14.11.3	The Single-Thread Rule / 单线程规则	951
Appendix / 附录		953

Graphics Programming

In this chapter

- 10.1 Introducing Swing, page 538
- 10.2 Creating a Frame, page 543
- 10.3 Positioning a Frame, page 546
- 10.4 Displaying Information in a Component, page 554
- 10.5 Working with 2D Shapes, page 560
- 10.6 Using Color, page 569
- 10.7 Using Special Fonts for Text, page 573
- 10.8 Displaying Images, page 582

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.

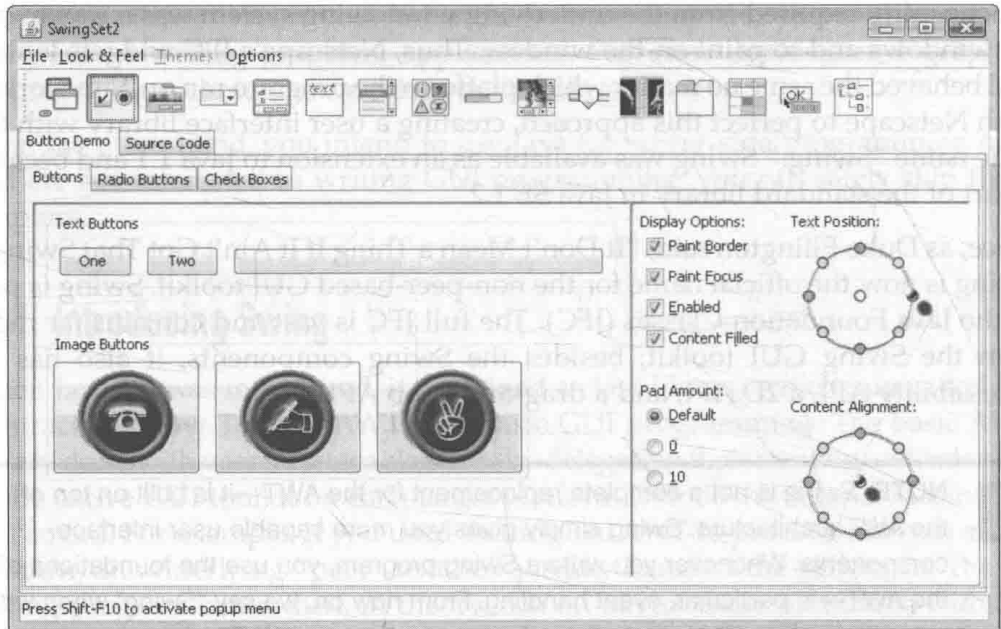


Figure 10.1 The Windows look-and-feel of Swing

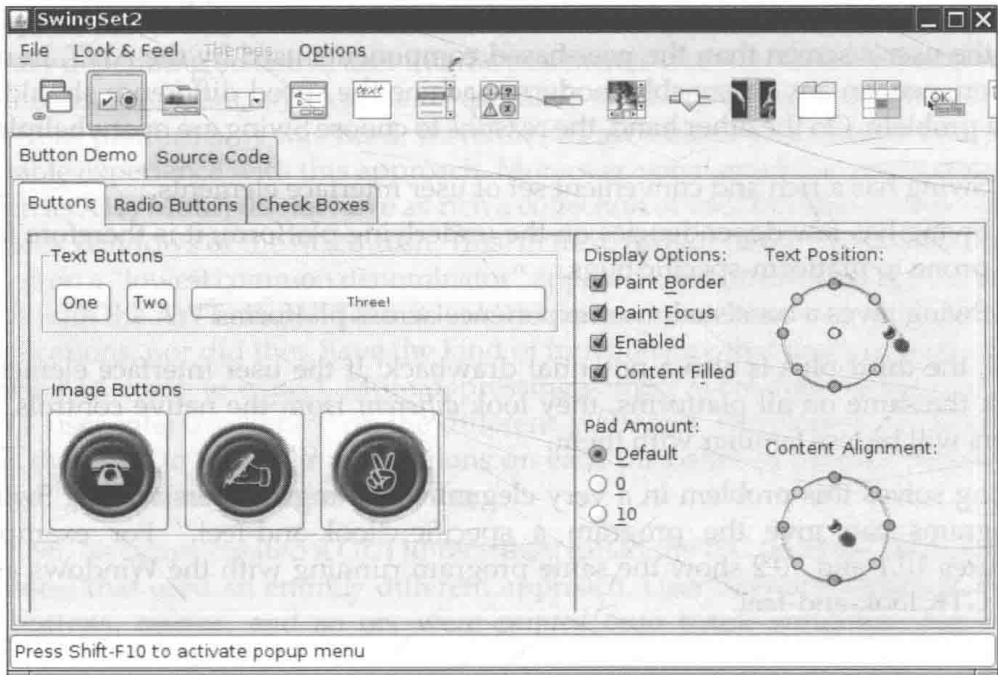


Figure 10.2 The GTK look-and-feel of Swing

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.”

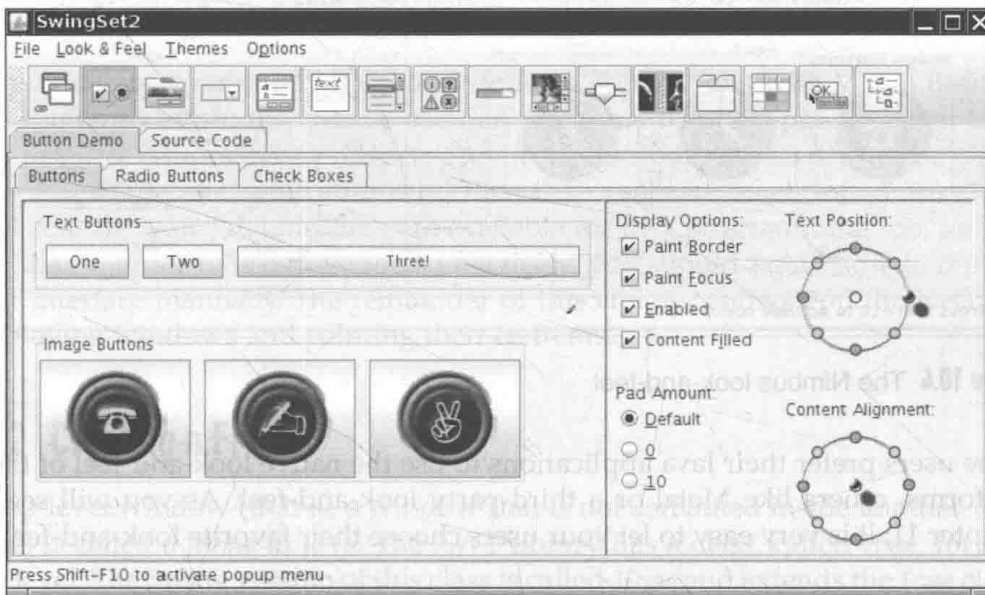


Figure 10.3 The Ocean theme of the Metal look-and-feel

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.

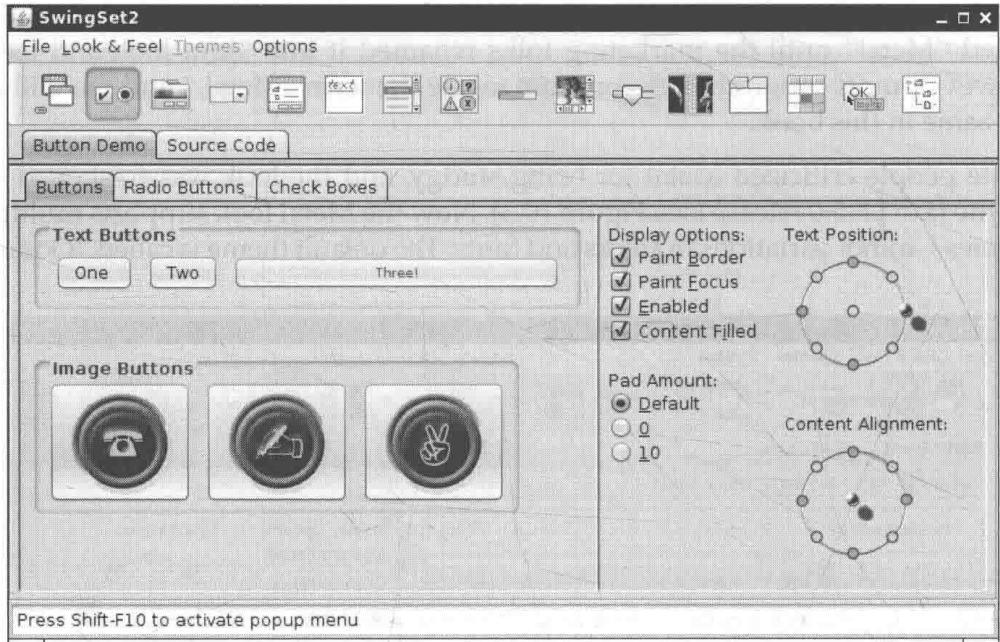


Figure 10.4 The Nimbus look-and-feel

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.javatoo.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.

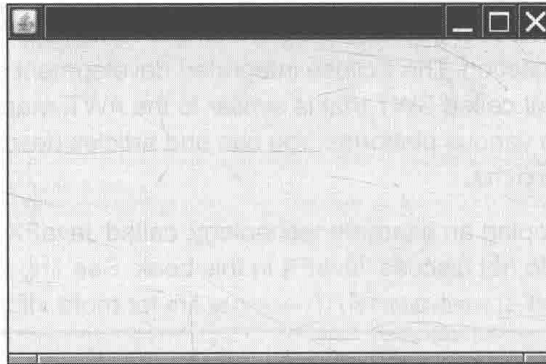


Figure 10.5 The simplest visible frame

Listing 10.1 simpleframe/SimpleFrameTest.java

```
1 package simpleFrame;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.33 2015-05-12
8  * @author Cay Horstmann
9  */
10 public class SimpleFrameTest
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater() ->
15         {
16             SimpleFrame frame = new SimpleFrame();
17             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18             frame.setVisible(true);
19         });
20     }
21 }
22
23 class SimpleFrame extends JFrame
24 {
25     private static final int DEFAULT_WIDTH = 300;
26     private static final int DEFAULT_HEIGHT = 200;
27
28     public SimpleFrame()
29     {
30         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31     }
32 }
```


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