

Introduction to the Java Foundation Classes

Topics in This Chapter

- 
- The History of the Java Foundation Classes
 - Overview of JFC Features
 - The Swing Packages
 - Look-and-Feel and the Model-View-Controller Architecture

Chapter

1

The Java Foundation Classes bring new capabilities to the Java programmer, foremost among which are the components in the so-called *Swing* set. This chapter begins by looking briefly at the history of the Java programming language and of the Abstract Window Toolkit in particular. The Abstract Window Toolkit, or AWT for short, provides the classes used to build an application's user interface. In both Java Development Kit (JDK) 1.0 and JDK 1.1, Java applications running under Windows looked just like Windows programs, while those running on Solaris looked the same as native applications written with the Motif toolkit. The reason for this was simply that much of the AWT is provided by code from the native platform's windowing system—the user interface components are rendered by Windows or by Motif, not by Java code. While this has its advantages, it also has drawbacks. For one thing, it is difficult to implement a single interface and map it to two (or more) host platforms that work differently. And even when you've done that, the controls themselves behave somewhat differently between the platforms. Because this behavior is part of the native windowing system, there's nothing you can do about it.

Implementing all of the user interface classes in Java gets rid of these problems at a stroke. That's exactly what the Swing components, which are the most significant part of the Java Foundation Classes (JFC), do—they replace the native implementation with a user interface library that works the same on all Java platforms.

This chapter starts by looking at the history of the Swing project and then moves on to look at the new architecture that was developed to implement the Swing controls. The power of this architecture will be shown toward the end of the chapter, when you'll see how simple it is to change the way an application built with Swing controls looks, without changing a single line of code.

What Are the Java Foundation Classes?

The Java Foundation Classes are a group of features whose implementation began with JDK 1.1 and is continued in JDK 1.2, one of the most significant parts of which, and the main concern of this book, is the Swing component set. The Swing components are all graphical user interface controls that replace most of the platform-native components provided by the JDK 1.0 and JDK 1.1 AWT. The best way to understand what the Swing components are and how they will affect the development of Java applications is to look at how the Swing project came about and at the problems with the AWT that Swing is intended to address.

In the Beginning: The Abstract Window Toolkit

Sun Microsystems released JDK 1.0 in the first half of 1996. A significant part of this new language was a package called `java.awt`, which contains the classes for the AWT.

In its early days, Java was associated very closely with the Internet and, thanks to its incorporation in Netscape's popular Web browser, the sight of cool applets written in Java became commonplace. A Web site without a Java applet of some kind soon became hard to find. While dedicated surfers searched for the next exciting applet, programmers were busy trying to meet user expectations using the facilities of the AWT—for it is the AWT that provides the user interface that, when viewed through a Web browser, becomes a Java applet.

Most of the original Java applets were of a similar, rather simple, type: moving images, dancing text, showers of pixels and just about anything that caught the eye. The aim was usually just to liven up an otherwise static Web page and attract as many callers as possible, in the hope that at least some of them might look at and, better still, buy the product or service that the Web site's owner was offering. Creating such simple applets did not really place much of a demand on the AWT. Most of the work revolved around loading and displaying sequences of images or animating some simple text. More sophisticated applets allowed user interaction using the mouse or the keyboard but, on the whole, the level of functionality required from the AWT by these applets was very low.

Alongside the applet developers, others were trying to use Java to develop weightier applications, such as office productivity tools and database client interfaces. While the applet developer was very happy with his new-found freedom of expression and got just about all he needed from the AWT, his colleagues were typically less impressed. For serious development work, AWT 1.0 simply did not stand up to scrutiny. For one thing, it was slow.

While this is not such an issue on the Internet, it assumes great importance to a user accustomed to applications written in C or C++ that usually (but with some notable exceptions) don't require you to wait a noticeable amount of time for some-

thing to happen after pressing a button. For another, the AWT simply wasn't robust enough—the implementation, particularly of the Windows version, was buggy and required developers to spend inordinate amounts of time looking for solutions to problems that weren't in their own code and having to produce work-arounds to patch up their applications and make them usable. Worst of all, though, the AWT didn't provide much variety in its range of user interface elements.

User interfaces have come a long way since the days of DOS. These days, even Unix has a windowing interface and only programmers and (some) system administrators still toil away at the shell prompt. Over the years, beginning with the introduction of Windows 3.0 and continuing up to the present day, users have become accustomed to interacting with applications using a mouse and a collection of familiar controls such as buttons, scroll bars, text fields and so on. Each release of Windows or the Motif toolkit brought new elements to the user's attention, most of which were quickly accepted and became indispensable.

Unfortunately, AWT 1.0 was not particularly sophisticated in its supply of visual controls, so developers who wanted to make their Java spreadsheet or word processor resemble existing products had to start virtually from scratch and write their own components.

Inevitably, this was a long and tiresome process, repeated in many companies around the world. The end result, of course, was that there never were that many serious Java applications developed with AWT 1.0 that made it to the marketplace, even in Beta form. By the time the development community had come to terms with creating their own components, JavaSoft had released JDK 1.1 and, along with it, version 1.1 of the AWT.

By comparison to its predecessor, AWT 1.1 was a great improvement. The Windows version was completely rewritten and was made much faster and more robust. Better integration with the user's desktop was provided and, for the first time, a Java programmer could give an application access to a printer without needing to write platform-dependent native code. The programming model was improved, too, with a better mechanism for handling events from the user interface and, with the introduction of the JavaBeans specification and its incorporation in JDK 1.1, it became possible for developers to create components that could be taken and reused elsewhere more easily and could even be incorporated into graphical application builder tools such as Microsoft Visual Basic or Borland JBuilder. But still, there remained the issue of user interface sophistication. Notwithstanding the breadth of JDK 1.1 and the immense improvement in the quality of the AWT, only two new components (a scrolling window and a pop-up menu) were added. Developers still had to produce most of their own user interface controls.

Enter Netscape

Meanwhile developers at Netscape had begun development of a set of improved user controls that were eventually released under the banner of the Internet Foundation

Classes (IFC). Implemented on top of AWT 1.0, the IFC components were a more complete set than their predecessors and included some nice features such as dialogs to allow a user to choose colors, files and fonts visually, buttons with images, sliders, bitmaps (images that load synchronously), support for animation sequences and an improved application framework that supported the nesting of parts of an application's interface inside other parts, with drawn boundaries to emphasize the nesting and grouping of components.

As well as providing improved functionality, the IFC components were different from AWT in another way. Whereas the AWT components are implemented partly in Java and partly as a platform-dependent native library, the IFC components are written entirely in Java. As a result, the IFC is immediately portable to any platform that supports Java and takes with it its look-and-feel, unlike the AWT, which adopts the appearance of its host platform.

The Swing Set: A Joint Effort

In the early part of 1997, developers from Netscape and JavaSoft began cooperating on a project that was dubbed Swing. The aim of this project was to bring together the AWT and the best parts of Netscape's IFC to produce a fully-featured, robust set of user interface classes to be released as part of JDK 1.2. Like their IFC counterparts, these components (referred to as the Swing component set) would be written entirely in Java to ensure portability and would, at some future time, allow JavaSoft to drop most of the peer model that the AWT had used for its first two releases. The peer model allowed JavaSoft to leverage native platform user interface support to get the first release of the JDK into the hands of developers very quickly. This approach had, of course, been extremely successful but was also, in part, responsible for the bad reputation of the AWT on the Windows platform.

The Swing project would soon grow far beyond a straight merge of the IFC components into the AWT, however. JavaSoft launched a *100% Pure Java* initiative, of which the Swing component set was, of course, a very good example and the number of components to be included in this set increased. The final result of this project was, as you'll see, a comprehensive new set of controls that matches the best of the competition and puts Java applications on a par with those written using the native platform libraries. Whereas the Java developer had been forced to limit the scope of the user interface or implement custom components, now the problem is more the wealth of choice available.

Swing Plus More: The Java Foundation Classes

The Swing components are, without doubt, the most notable part of what JavaSoft named the Java Foundation Classes, but they are not the whole story. Several improvements that had gone into AWT 1.1, plus a few more enhancements planned

for the JDK 1.2 time frame, were brought together and placed under the JFC banner. In total, the JFC set consists of the following pieces:

- The Swing components
- The Desktop Colors feature of JDK 1.1
- The JDK 1.1 Printing facility
- The Java2D API, which supports enhanced text, color and image support
- Accessibility, which provides support for technologies that make it easier for users with disabilities to use the Java platform
- The JDK 1.1 cut-and-paste and clipboard facility, combined with a new drag-and-drop facility introduced in JDK 1.2.

The full range of JFC features is delivered as part of JDK 1.2, but JavaSoft also provides a package, called JFC 1.1 (also known as Swing 1.0), that contains the Swing components in a form suitable for use with JDK 1.1. This package allows developers to make use of the Swing components without having to wait for the final release of JDK 1.2. This book focuses on the parts of the Java Foundation Classes that are found in both the add-on package and in JDK 1.2. In addition, it covers the JDK 1.1 printing facility and a host of other features from JDK 1.1 that are closely related to the Swing components, all of which must be properly understood in order for you to make full use of Swing. This subset of the full JFC family consists mainly of the first three items just listed.

What Is the Relationship between JFC and AWT?

With the introduction of a new set of components that use a different software architecture from those already in the AWT, it is natural to wonder what will now become of the AWT itself. As early as the last Beta version of JDK 1.1, JavaSoft were signaling that there was a major change on the way for the AWT. In this last release before final customer shipment, the AWT engineers introduced a new facility called Lightweight Components. On the surface, this simply represented the ability to produce a component or a container with a transparent background. This was done by allowing developers to directly extend the `Component` and `Container` classes and a nice example of a button with rounded edges was provided in the release documentation. However, also among the documentation was a paper entitled *AWT: The Next Generation*. This paper indicated that the future of the AWT lay in the direction of lightweight components, enabling the removal of most of the troublesome peer model. It also indicated that the peer model would be retained for a while for compatibility reasons. This paper, of course, was the first public announcement of the initiative that would eventually result in the delivery of the Swing components.

In JFC 1.1 and JDK 1.2, the old AWT components and the Swing components are both supported and the AWT components continue to use the peer model. It is not clear how long this situation will continue. Some of the Swing components are actually derived from an AWT component—for example, the `JFrame` component, which provides the top-level window for an application, is derived from the AWT `Frame` class. However, every AWT component has a corresponding Swing component, so it is already possible to implement an application that does not directly depend on the AWT, apart from the top-level window.

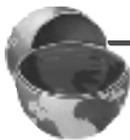


Core Tip

Given the obvious desire on the part of JavaSoft to move away from the peer model, developers would be well advised to upgrade existing applications as soon as possible, to remove future dependence on the old AWT components.

Because the AWT infrastructure is still in place, applications written with the old AWT continue to work in JFC 1.1 and with JDK 1.2 in the same way as those using the JDK 1.0 event model continued to work in JDK 1.1. Because of the clean separation of the two sets of components and the fact that they are all ultimately derived from the `Component` class and share the same event model, it is possible to mix AWT and Swing components in a single application. Of course, it is extremely unlikely that anybody would develop a new application that relied on both sets of components, but the ability to mix allows developers to migrate from the old set to the new incrementally, retaining a working application throughout.

This feature is of enormous importance if you have spent great efforts developing your own custom components based on the AWT classes. Most importantly, it means you can continue to use them until you create Swing-based replacements. However, because of the breadth of coverage of the Swing components, it is very likely that you will be able to use them to directly replace a large proportion of your custom controls, avoiding the need to carry out any porting at all.



Core Note

You will often see references in this book to AWT components and Swing components as if they were mutually exclusive sets. Strictly speaking, this is not true, because all of the Swing components are also AWT components. What we really mean when we say AWT components is the set of components that the AWT provided in JDK 1.1.

What Do I Need to Relearn to Use the Swing Components?

The simple answer to this question is that you should be able to make basic use of some Swing components straight away. Swing does not fundamentally change the way in which Java applications are constructed. You still create a top-level window; you still use frames, components and layout managers; and you still connect them together in almost the same way as you always have (but see the discussion of `JFrame` in Chapter 2, “Frames, Labels, and Buttons,” for an important exception). The main problem you have to overcome to use Swing proficiently is being aware of all the possibilities available to you. Instead of the handful of AWT components, each of which was very simple and required little customization, you now have a very large and very rich set of possibilities to choose from. In addition to the wide choice, many of the components are very highly customizable, especially if you are prepared to spend time implementing some of the pluggable *helper* classes that can be attached to some of them. The aim of this book is to help you to do that.

As an example, consider the new Swing Combo box. This control allows a user to select from a list of possible values and shows the selected value in an input field. To see the list of possibilities, you click on a small arrow near the input control to reveal a drop-down list. If the programmer has made the Combo editable and the value you want is not in the list, or if you know the value without needing to refer to the list, you can type it directly into the input field. That’s the basic functionality of the control.

However, if you are prepared to do a little work, you can provide your own way of rendering the contents of the drop-down box or of editing the input field. You don’t have to restrict yourself any more to a traditional drop-down combo box containing a list of strings. For one thing, without doing very much, you can add images to the strings. Figure 1-1, for example, shows three combo boxes, one of which only uses images.

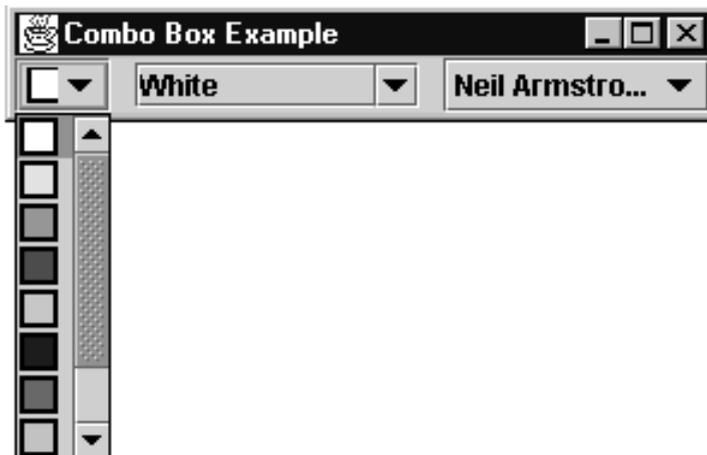


Figure 1-1 Swing combo boxes.

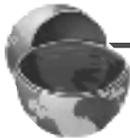
With slightly more effort, you can present a hierarchical view of a file system from which you could select a file to operate on. Or maybe you would prefer a drop-down scientific calculator, or, in a financial institution, a way of choosing from a selection of pricing information held on a server somewhere.

Overview of JFC Features

The JFC provides a very wide range of components and facilities. Before we start our detailed investigation, we'll spend a short time in this section looking briefly at the various parts of the Java Foundation Class API that fall within the scope of this book.

The JComponent Class

The `JComponent` class is the common superclass of almost all of the new Swing components. In the same way that `Component` provides common methods and states for all AWT components, `JComponent` performs several major functions for its own subclasses, several of which will be briefly described in the next few sections. `JComponent` is derived from the AWT `Container` class, which is itself a subclass of `Component`. Because `JComponent` extends `Container`, it is a lightweight object. It does not have a window in the native windowing system like the AWT components do and it can be transparent. The other advantage of extending `Container` is that a `JComponent` can be constructed from many different components, because it has built-in container functionality.



Core Note

It is important to understand the relationship between `JComponent`, `Container` and `Component` and between `JComponent` and all of the other Swing classes. Appendix A contains a class hierarchy diagram that shows every component in the Swing set. It's a good idea to refer to this diagram as each new component is introduced so that you can see where it fits in the overall picture.

`JComponent` enables the pluggable look-and-feel facility that will be discussed later in this chapter, by redirecting calls that would otherwise be handled by `Component` methods to a separate user interface object that is responsible for maintaining the component's appearance. For example, `JComponent` overrides the `paint` method and redirects it to the user interface component instead of allowing `Component` to process it. This issue is discussed further in "The Model/View/Controller Architecture" below and in Chapter 13, "The Pluggable Look-and-Feel." Many other Swing features are also based on support provided by the `JComponent` class.

Frames and Dialogs

In Swing, the top-level windows have changed. Instead of `Frame`, there is `JFrame` and, similarly, there are `JWindow` and `JDialog` classes. The difference between these Swing components and their AWT counterparts is much more than just a name change. Whereas the AWT components were straightforward containers that were special only because their native windowing system peers were top-level windows, the Swing components have a complicated internal structure that is visible to the programmer and that allows them to be much more useful when constructing user interfaces.

As an example of this, all of these containers can support a menu bar, a facility that was previously only available with a `Frame`. In addition, the main working area can be treated as if it had many independent layers. Placing components in different layers makes it possible to arrange for them to overlap, and also to ensure that some components always appear in front of others. This facility can be used to great effect to provide support for multiple document applications, which is covered in Chapter 12, “Multiple-Document Applications.”

The working area of a `JFrame` can be divided in two areas using the Swing `JSplitPane` control. This component allows the user to drag a moving boundary either horizontally or vertically to adjust the space available to two components mounted on the frame. This control is useful when creating layouts like that of the Microsoft Windows Explorer, or the network news interface provided by some Web browsers, where the list of news groups and the list of items in the current news group are displayed side-by-side with a slider that allows the space available to one list to be increased at the expense of the other. `JSplitPane` is covered in Chapter 9, “Text Components,” where it is used to show two different views of a single document.

Swing also provides a gamut of new support for dialogs, which were very hardly catered for at all in JDK 1.1. The old `FileDialog`, which was very limiting and fell far short of similar facilities on the native platform, has been replaced by the more powerful `JFileChooser`, which can be used on its own or as part of a larger dialog. Other extended dialog support includes `JColorChooser` for graphically selecting colors and a host of message, warning, error and information panes provided by the `JOptionPane` class.

One of the most impressive-looking Swing controls is the `JTabbedPane`, which looks and behaves like a Windows property sheet. `JTabbedPane` is especially useful in dialog boxes, where it allows the programmer to create panels of controls that control related parts of an application's configuration and present them in an uncluttered manner, separate from configuration information for other aspects of the application, while still having all of a program's configurable options available in one place.

The wide range of dialog components is discussed in Chapter 7, “Using Standard Dialogs,” and Chapter 8, “Creating Custom Dialogs.”

Per-Component Borders

All of the Swing components provide for the drawing of a border around their edges. Borders are managed by the following `JComponent` methods:

- `public void setBorder(Border b);`
- `public Border getBorder();`

`Border` is an interface, not a class, and the Swing set contains several standard borders that should meet most requirements.

A border can be used to group together controls that are related to each other and don't need to be closely associated with other controls on the same panel. For example, it might be appropriate to surround a group of radio buttons with a border that contains some text to describe what the buttons control, as shown in Figure 1-2. Alternatively, you can use the `setBorder` method to replace the border of a standard component, such as a text input field, with one of your own choice.



Figure 1-2 A Swing titled border grouping three radio buttons.

Borders are discussed in Chapter 4, “Graphics, Text Handling and Printing,” and in Chapter 8.

Graphics Debugging

Often it can be difficult to see why complicated layouts or graphics are not being rendered properly, because the entire process happens so quickly that it is impossible to see exactly what is being done. Alternatively, you may find that sometimes your layouts are being redrawn too frequently. In both of these cases, it would be useful to

have a way to either slow down the rendering process or to have a record of what was done so that it can be inspected later for redundant operations. JFC 1.1 includes a new Graphics Debugging facility that provides both of these features and `JComponent` provides the interface to it via the `getDebugGraphicsOptions` and `setDebugGraphicsOptions` methods. This feature is covered in detail in Chapter 3, “Managing the User Interface.”

Enhanced Mouseless Operation

Swing provides an improved mechanism for allowing an application to be driven from the keyboard as well as using a mouse. JDK 1.1 introduced keyboard accelerators for menu items and a better mechanism for managing focus traversal using the `Tab` and `Shift-Tab` keys. Swing extends this by allowing actions to be triggered by particular key sequences on arbitrary components on the user interface. This new mechanism makes it almost trivial to support function keys and other types of hot keys that were very difficult to implement with JDK 1.1. `JComponent` provides the repository for the configuration information for this mechanism. The Swing components also provide a more flexible focus management model, including the ability to install a customized focus manager. Accelerator keys and focus management are discussed in Chapter 5, “Keyboard Handling, Actions and Scrolling.”

Tooltips

A nice feature of some user interfaces is the ability to show a small help window or *tip* when the mouse pauses over a button. Typically, this window would contain text that describes what the button would do if it were pressed. Swing generalizes this mechanism by making it available in all of the new components.

Simple applications can make use of the facility by supplying some text that will display in the “tip” window, as shown in Figure 1-3, by invoking the `JComponent` `setToolTipText` method. If you want to be a bit more clever, you can take control of this mechanism by providing your own component to be used instead of the default window with text, or you can arrange for the text that is shown to be dependent upon the position of the mouse relative to the control. We cover this mechanism, and show how to exploit it to the full in Chapter 8.

Enhanced Scrolling

In JDK 1.1, some of the AWT components supplied scroll bars if the information that they needed to display did not fit in the available screen space. Programmers could also create their own scrolling components using the primitive `Scrollbar` or the `ScrollPane` container, which handles most of the details of scrolling for the simplest cases.



Figure 1-3 A tool tip.

By contrast, Swing components do not provide their own scroll bars—if, for example, the text in a text area could turn out to be too large to be seen on the screen at once, it is up to the programmer to provide the scrolling functionality. Fortunately, the Swing component set provides a very simple to use but extremely powerful scrolling container, `JScrollPane`, that fully replaces the AWT `ScrollPane` and can be added to any Swing component with only one line of code.

When a component is wrapped with scroll bars, it is often useful to force a particular part of the scrolled area to become visible. As an example of this, consider the case of a text control that provides a search facility. As the search progresses through the text, it is necessary to scroll the content so that matched parts of the text are in the visible region. `JComponent` provides a method that can be used to request that a scrolling parent object change its viewport to make some part of the calling component visible. Similarly, if the user drags an object over a scrolled list, `JComponent` provides the means to make the list scroll automatically, without intervention by the list itself, so that the position in which the user might want to drop the component becomes visible. Compare this to dragging files between directories in the Windows Explorer, for example. If the target directory in which the file is to be dropped is not visible, dragging the file to the top or bottom of the window that is showing the directory tree causes that window to scroll in the appropriate direction. This mechanism and the Swing scrolling controls are described in Chapter 5.

Pluggable Look-and-Feel

Without doubt one of the more interesting and novel features of the Swing architecture is the fact that the applications it produces have a platform-independent look-

and-feel to them, because the user interface is rendered not by Windows or the Motif library, but by Java code that will work the same way on every platform. As a result, you can take an application developed using Swing components on Solaris, say, and have it run with the same appearance on Windows 95. But that's only half the story.

Unlike the old AWT, the parts of the Swing classes that deal with drawing components onto the screen are not an inseparable part of each component. Instead, each control delegates its screen drawing to a separate entity that knows how to draw components of that type. For example, a button object allows a separate button-drawing class to render its image onto the screen; the button itself would be concerned only with delivering a notification to the application program that it has been pressed.

Once you have separated the rendering of components from the components themselves, it becomes possible to substitute a different rendering class that draws the button in a different way and what you can do for a button you can also do for every other interface component. From this idea comes the concept of a family of user interface classes that implement a consistent look-and-feel across all of the components. The Swing components as supplied in JFC 1.1 and JDK 1.2 come with several look-and-feel implementations, among them two that emulate the Windows and Motif look-and-feel and a third that is a cross-platform look-and-feel specifically designed for Java applications, called the *Metal* look-and-feel. All the user has to do is *plug* the appropriate look-and-feel set into the application by configuring a default for his or her platform. You'll see more on this later in this chapter and in much greater detail in Chapter 13.

Core Note

There are at least two other look-and-feel implementations available from JavaSoft but not included in the standard Swing release. One of these is a Mac look-and-feel that gives Apple Macintosh-like behavior to the application. The other is another cross-platform look-and-feel called Organic, which was originally available in the later developer prerelease versions of Swing under the name of the Java look-and-feel.



Layout Managers

Swing adds two new layout managers to those provided by the AWT. `BoxLayout` is a useful layout manager that arranges its components in either a single row or a single column. As such, it is ideal for managing groups of buttons or other components that have to remain properly aligned in one direction. It also provides the ability for the programmer to specify how its components should move and resize when the container is expanded, how much of the extra space should be used to let the components grow and how much should be left empty. The `OverlayLayout` manager can be used to arrange for components to overlap each other and to stay overlapped

as their container expands. `OverlayLayout` can be used in conjunction with transparent, lightweight components to build up an interface from several layers, with each layer being composed of a different component.

Because layout managers have generally been poorly documented and a proper understanding of them can save a lot of time and effort when developing applications, this book provides an extended description of the complete set of layout managers, both the old AWT ones and the Swing ones, in Chapter 3.

Labels and Buttons

The AWT `Label` and `Button` classes were very simple and offered limited functionality. By contrast, the Swing `JLabel` and `JButton` classes are highly customizable. Labels and buttons can have both text and an image associated with them and it is possible to choose the relative positions of these items and their overall alignment in relation to the control itself. The Swing button classes are, in fact, a hierarchy that includes the Swing menu items, traditional push buttons, toggle buttons that are “sticky” (that is, they remain pressed in until pressed again), check boxes and radio buttons. All of these types of buttons share the features of `JButton`, including the ability to present an image. Among the possibilities that this opens up is the potential to represent a check box or a radio box with an image more in keeping with the application it is being used in than the default square box or the circled dot that are traditionally used. You’ll see all of this functionality, and more, in Chapter 2, “Frames, Labels, and Buttons.”

Menus and Toolbars

The AWT menu system was very restricted and idiosyncratic. Menus were restricted to frames, where they had to be placed directly under the caption bar and above the useful working area. Furthermore, menus and menu items were not derived from the `Component` class, which meant that they often couldn’t be treated in the same way as the other components in an application. JDK 1.1 added a menu shortcut facility, but even that was disappointing because it could be used with menu items, not menus.

By contrast, the Swing menus are all derived from `JComponent` and they are all implemented entirely in Java. As a result, they behave predictably across all platforms and they don’t exhibit any platform-specific peculiarities or limitations. AWT-style menu shortcuts are supported on both menus and menu items and, in addition, it is possible to attach *mnemonics* that allow a menu item to be activated with a single keystroke when it is visible and *hotkeys* that activate the menu item even when it is not visible. Figure 1-4 shows a selection of menus and menu items, with mnemonics indicated by underlines.

Swing menus also have other features that are taken for granted elsewhere. For example, you can add an image to the text on a menu item, or remove the text altogether and let the image stand alone, or you can change the font and color of the text or its background subject, of course, to constraints placed by the look-and-feel that

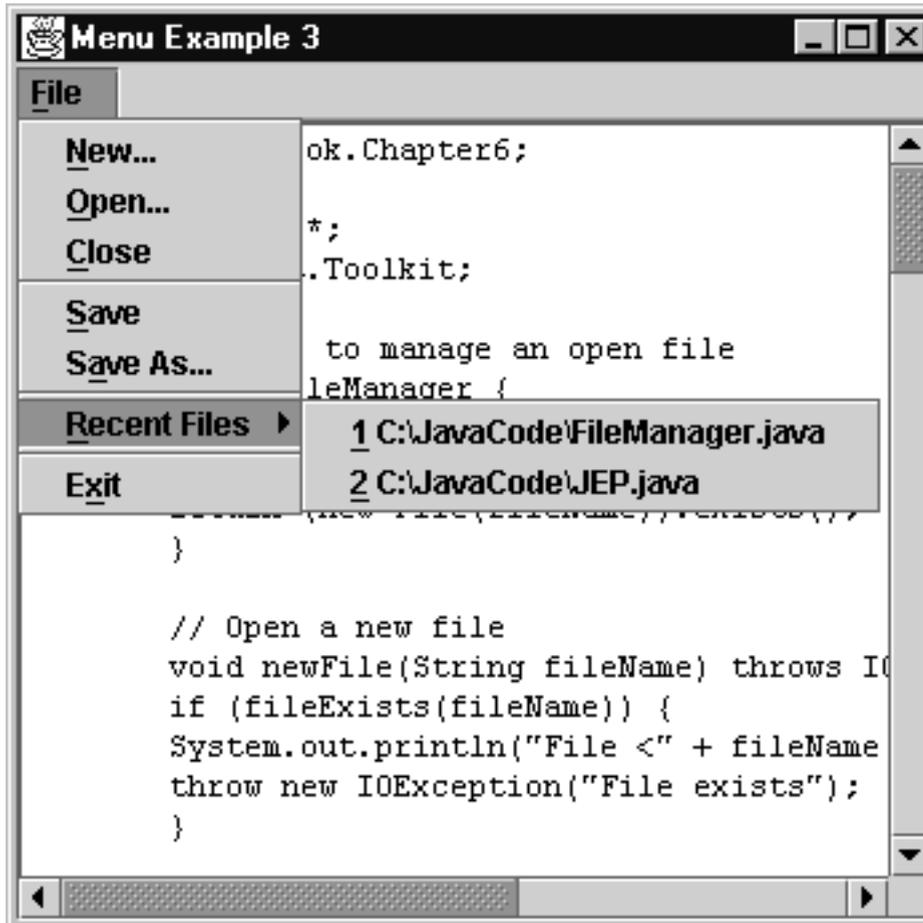


Figure 1-4 Swing menus and menu items.

the user has selected. For the benefit of the programmer, menus now post events as they are posted and removed from the screen so that they can be created and changed in a lazy manner, as they can in other windowing environments.

A relative of the menu bar is the Swing toolbar. The toolbar hosts a two-dimensional arrangement of components, usually buttons with mnemonic images, that allow very fast access to important features of the application. A typical toolbar is shown in Figure 1-5. The toolbar will usually be located under an application's menu bar but it can be placed anywhere on the application's window. Toolbars can also be configured so that they can be completely detached from the window, to float in a separate frame and can later be redocked with the frame under program or user control.

Menus and toolbars are discussed in Chapter 6, "Menus and Toolbars."

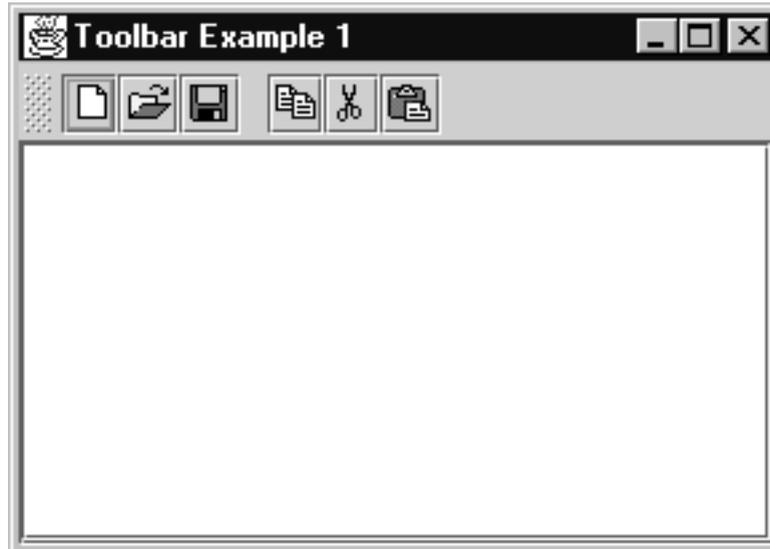


Figure 1-5 The Swing Toolbar.

Text Controls

Swing provides lightweight replacements for the AWT `TextField` and `TextArea` controls that provide all the functionality of their predecessors, including the ability to detect and track changes to their content as they are made. The Swing components are, however, built on a complex infrastructure that makes it possible to create more complex text controls that can render text in multiple fonts and colors, possibilities that the AWT does not offer. Swing includes the `JTextPane` control, which supports text in various styles and can intermix text with images and even AWT and Swing components, to create documents with embedded functionality, and `JEditorPane`, which can display documents encoded in many different input formats, including HTML. Chapter 9 contains a detailed look at the Swing text components and the underlying support that makes their powerful functionality possible.

Data Selection and Display Controls

One area in which Swing is much more complete than the AWT is its provision of controls for selecting and displaying data. Where the AWT provided the `List` and `Choice` controls, Swing has `JList` and `JComboBox`, which do everything that their predecessors do and more. Both of these Swing controls can handle large amounts of data easily, whereas on some platforms the AWT components do not cope well with large lists. The Swing controls can also be customized to represent the data

they contain in various ways, including the use of images as well as or instead of text and they can also hold selections that are not limited to the strings that the AWT components require. These components are both described in Chapter 8.

Swing also has two data display controls that have no precedent in the AWT. `JTree` is a very flexible control for displaying data organized in a hierarchical form. As such, it can be used to display, for example, a graphical representation of a file system or anything else with a similar structure. `JTable` is used to display data that is organized in two-dimensional row and column form and therefore is a natural choice for representing data returned from queries made to a database. Among the facilities of `JTable` is the ability for the user to rearrange and resize the columns in the table. Both `JTable` and `JTree` can be customized in various different ways and they can both allow the user to edit the data being displayed.

Core Note

Both `JTree` and `JComboBox` merit chapters of their own (they are described in Chapter 10, “The Tree Control,” and Chapter 11, “The Table Control,” respectively).



Timers

JDK 1.1 had no support for timers. If you needed one, the best you could do was to create a separate thread that slept for the required time and then resumed to perform a delayed action or to post an event to another thread. Swing provides the `Timer` class that allows you to create either a one-off timer that fires once and then stops, or a repeating heartbeat timer. We look at timers and see an example of their use in Chapter 8.

Support for Applets

You can use Swing classes in applets in the same way as you can use AWT classes. In fact, there is very little that we will say about Swing in this book that doesn't apply equally to applications and applets. For this reason, you won't find any applet-specific examples in this book—there simply is no need to make the distinction between these two environments with Swing any more than there is with AWT. There are, however, a couple of points to know about Swing and applets.

First, Swing applets must be based on the `JApplet` class instead of `Applet`. `JApplet` is, in fact, a subclass of `Applet`. Secondly, `JApplet` has the same internal structures as the `JFrame` class that you'll see in Chapter 2. This means that you can have an applet with a layered display and even a menu bar if you want one. Of course, the security restrictions that apply to applets in JDK 1.1 still apply when you use Swing classes in applets, although there are changes in JDK 1.2 that make it possible to relax some of the restrictions under certain circumstances. This is not, however, a Swing issue.

The Swing Packages

The JFC 1.1 product is an add-on to JDK 1.1 that contains the Swing components in a collection of packages, the names of which all start with `com.sun.java.swing`. In JDK 1.2, the same packages exist, but the package name has been changed to `java.awt.swing`. In the code shown in this book, the Swing packages are always assumed to be at `com.sun.java.swing`, etc, which is appropriate for those using JDK 1.1. However, all of the examples are included twice on the CD-ROM, so if you intend to use JDK 1.2, you can install a suitable set of source and class files.

The reason for this duality is to allow applets and applications to be written using the Swing components before the final release of JDK 1.2, or by those who don't want to (or can't) move to JDK 1.2 immediately. The alternative was to place the new components in the `java.awt.swing` package straight away and issue this as an add-on to JDK 1.1, to be naturally superceded by JDK 1.2. However, for security reasons, packages starting with `java` should not be downloadable to browsers. Since this means it would not be possible for the owner of an applet to have the browser download both the Swing-based applet and the Swing classes (in a JAR file), the only alternative would be to require anybody who wanted to view a Swing-enabled applet to obtain and install the `java.awt.swing` packages themselves. Obviously, this is not an inviting prospect for applet writers, since a significant number of their potential customers would not be inclined to do this and therefore the applet's impact would be correspondingly reduced.

By issuing the Swing components in a package structure that is outside the core Java hierarchy, it is possible to have the browser download the Swing JAR along with the applet, or for the applet supplier to create a subset of the Swing JAR that contains only the Swing facilities that the applet requires and have that downloaded to minimize startup time.

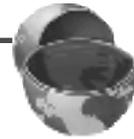
Here are the packages that are common to JFC 1.1 and JDK 1.2, grouped by functionality.

- `com.sun.java.swing`
- `com.sun.java.swing.border`
- `com.sun.java.swing.event`
- `com.sun.java.swing.plaf`
- `com.sun.java.swing.plaf.basic`
- `com.sun.java.swing.plaf.metal`
- `com.sun.java.swing.plaf.motif`
- `com.sun.java.swing.plaf.multi`
- `com.sun.java.swing.plaf.windows`
- `com.sun.java.swing.preview`

- `com.sun.java.swing.table`
- `com.sun.java.swing.text`
- `com.sun.java.swing.text.html`
- `com.sun.java.swing.text.rtf`
- `com.sun.java.swing.tree`
- `com.sun.java.swing.undo`

Core Note

The Swing release also contains a package called `com.sun.java.accessibility` that provides support for Java Accessibility in the Swing components. Although the Swing components in JFC 1.1 and JDK 1.2 do implement this support, Accessibility is not covered in this book, so does not appear in this package list.



`com.sun.java.swing` and `com.sun.java.swing.preview`

This package contains the Swing components themselves and many of the interfaces that they use. The classes and interfaces in this package follow a naming convention that helps to identify what type of object they are. For example, the GUI components themselves have names that begin with a `J` (although `Box` is an exception to this rule, because it is not derived from `JComponent`); there is a Swing component to replace every AWT component and usually you can deduce its name by just adding the `J` prefix. A notable exception to this rule is `Choice`, which is replaced by `JList`, a Swing component that also supercedes the AWT `List` control.

Many GUI components and other classes are closely related to each other and have much of their code in a shared base class; these shared classes all have names that start with `Abstract`. Examples of this are `AbstractButton`, which is the base class for all of the Swing buttons (and even for menu items) and `AbstractAction`, which is a basic implementation of the `Action` interface that will be introduced in Chapter 5. As their names suggest, these classes are all abstract.

Many Swing components are composed of several parts; a typical component is made up of a class that represents the control itself (such as `JButton`), a class that knows how to draw the component on the screen and another class that represents the state of the component, known as the component's *model*. The Swing package contains several interfaces that define the methods that the model provides as well as actual implementations of those interfaces that are used by real components. Buttons, for instance, have a model that implements the `ButtonModel` interface. The actual implementation of this model that all of the Swing buttons use is in a class called `DefaultButtonModel`. This naming scheme, whereby a basic, but com-

plete, implementation of an interface is placed in a class whose name is that of the interface with the added prefix `Default`, extends to the other model interfaces and their implementations in the Swing package and is also used in some of the other packages.

The Swing package also contains some classes that are used in the implementation of new mechanisms that can be used with the components themselves. For example, Swing introduces a more flexible focus management mechanism that allows much finer tuning than the mechanism provided in JDK 1.1. The basic methods that make up a Swing focus manager are contained in the abstract `FocusManager` class and there is a complete implementation of a specific focus management policy in the class `DefaultFocusManager`. Similarly, there are classes that support the new API for managing keyboard accelerators and the pluggable look-and-feel mechanism that will be described later in this chapter.

As the deadline for the first official Swing release approached, JavaSoft moved some components from the Swing package into a preview package, reflecting the fact that their API was not yet stable. This package is called `com.sun.java.swing.preview`. If you are going to be making use of Swing with JDK 1.1, bear in mind that the programming interface of the components in the preview package is very likely to change. This could cause compatibility problems if you plan to move to JDK 1.2 later.

`com.sun.java.swing.table` and `com.sun.java.swing.tree`

Swing provides two powerful components that allow you to present collections of data in the form of a tree or a table. The classes for the components themselves, `JTree` and `JTable`, reside with the other GUI components in the Swing package, but both components are sufficiently complex that their data model classes and the other helper classes that are required to make them useful are held in separate packages to avoid cluttering the Swing package and to make it easier to see what is available from `JTree` and `JTable` themselves. Like many of the Swing components, these controls are highly configurable and, by replacing or extending the *renderers* that draw parts of their screen representation, you can fundamentally change the way these controls look. The renderer interfaces are held alongside the tree and table support classes in the `com.sun.java.swing.tree` or `com.sun.java.swing.table` package as appropriate, while the default implementations are look-and-feel specific and so reside in the look-and-feel packages.

The Text Packages

The Swing text components are much more complex than those in the AWT. While you can regard the relatively simple `JTextField` and `JTextArea` controls as straightforward replacements for `TextField` and `TextArea`, in implementation terms they are very different. All of the text controls are derived from the base class

`JTextComponent`, which resides in the `com.sun.java.swing.text` package. `JTextComponent` itself is only a generic wrapper for the large collection of classes that keep track of the content of a text control and how it should be rendered when displayed, the classes that actually display the text and those that react to user input from the keyboard or the mouse. All of these classes reside in the `com.sun.java.swing.text` package.

The Swing package itself contains three other text components—`JPasswordField`, `JTextPane` and `JEditorPane`, which all rely on support from the classes in the `text` package. In addition to this, `JEditorPane` can be configured to render text stored in various different forms, including HTML and Rich Text Format (RTF), for which it uses classes in the `com.sun.java.swing.text.html` and `com.sun.java.swing.text.rtf` packages.

A common requirement in sophisticated text management applications, such as word processors, is to be able to undo changes made to the text. The text components are all implemented in such a way that changes made to the control's data model are recorded as transactions that can be reversed (in the right order) or re-applied after being reversed (from the right initial state), so that it is possible to expose to the user an undo/redo facility. The `com.sun.java.swing.undo` package contains classes that work with the information provided by the text components to make it easier to provide this support.

`com.sun.java.swing.border`

This package contains all of the standard borders that Swing provides, together with the `Border` interface that borders must implement, and an abstract base class, `AbstractBorder`, that forms the basis of all of the Swing borders and that can be used to create new ones.

As you'll see in Chapter 8, borders are not usually created directly. Instead, so that border instances can be shared between components whenever possible, a new border is usually created by using the `BorderFactory` class, which arranges to satisfy a request for a new border by returning an existing instance if it can be shared. `BorderFactory` resides in the Swing package.

`com.sun.java.swing.event`

The Swing components bring with them many new events and event listener classes. Just as the usual AWT events are all held in the `java.awt.event` package, the Swing events and their listeners reside in `com.sun.java.swing.event`. Among the new events are `ChangeEvent`, which reports an unspecified change of state in its source (see the discussion of progress bars and sliders for example of this event) and `TreeModelEvent` that is generated when the content of a tree's data model changes. While some of the events, such as `ChangeEvent`, extend the existing

`AWTEvent` that is the basis of the events in `java.awt.event`, the majority of them, like `TreeModelEvent` are not component-based and are therefore derived directly from `java.util.EventObject` instead. This reflects the fact that most of the events in this package represent things that happen inside a component or as a result of some operation performed by the component on itself, possibly as an indirect result of user interaction, rather than arising directly from actions at the user interface. Changes to the tree's data content can, for example, occur if the user is allowed to edit the tree contents, but the source of the event is the tree's data model itself reporting that it has been changed rather than the tree component, which is just the visual representation of the data.

`com.sun.java.swing.plaf`

As you already know, a major feature of the Swing components is their ability to be rendered in different ways depending on the look-and-feel packages that are installed on a particular system. This is made possible by placing all of the code that knows how to draw the component in a separate class from the one that the application interacts with, so that it can be changed at run time without affecting code in the application that is holding references to the component objects themselves. In order for this to work, there must be a well-defined interface between each component type and the class that implements its look-and-feel. The `com.sun.java.swing.plaf` contains all of these interfaces (`plaf` stands for Pluggable Look-And-Feel).

All of the objects in this package are actually abstract classes that both specify the actual look-and-feel interface and, in some cases, contain prototypical implementations of some of it. All of them are derived from `com.sun.java.swing.plaf.ComponentUI`, which represents the generic interface (or minimum contract) between a component and its user interface class. The amount of usable code in these classes that could form part of a real look-and-feel class implementation varies from class to class.

This package also contains classes that wrap default values stored by look-and-feel implementations that are used by the user interface classes. For example, many components have several associated colors that are used to fill part of their screen representation. The colors that a control uses will, of course, depend on the look-and-feel, so they are stored separately by each look-and-feel. Instead of storing the colors as a `java.awt.Color` object, however, the color is wrapped with an instance of `com.sun.java.swing.plaf.ColorUIResource` and there are similar classes to wrap fonts and other resources. In Chapter 13, you'll see why these wrapper classes are used.

It is important to realize that this package does not contain any actual look-and-feel implementation: it just stores the classes that define the interfaces that these implementations use.

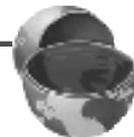
The Look-and-Feel Packages

Several packages provide the user interface classes for the look-and-feel implementations supplied with Swing. All of these classes reside below `com.sun.java.swing.plaf`. The `com.sun.java.swing.plaf.motif` package, for example, contains classes that know how to render all of the Swing components and react to mouse, keyboard and focus changes in such a way as to make the components look and feel as if they were part of a Motif desktop application. Similarly, the `com.sun.java.swing.plaf.windows` package provides the Windows 95 and Windows NT 4.0 look-and-feel, while the `com.sun.java.swing.plaf.metal` package contains a custom look-and-feel designed specifically by JavaSoft for Java applications that need to look the same on all platforms.

The `com.sun.java.swing.plaf.basic` package does not provide a look-and-feel implementation that the user can elect to use. Instead, it provides a set of user interface classes, one for each Swing component, that can be used (by programmers) either directly or as the basis for a more customized one in a real look-and-feel package. For example, the user interface class for the `JTree` component, `com.sun.java.swing.plaf.basic.BasicTreeUI`, is used as the base class for the tree user interface for Windows (`WindowsTreeUI` in the `com.sun.java.swing.plaf.windows` package), Motif (`MotifTreeUI` in `com.sun.java.swing.plaf.motif`) and for the Metal look-and-feel (`MetalTreeUI` in `com.sun.java.swing.plaf.metal`). You'll see examples of the available user interfaces and how they render various components later in this chapter and throughout this book.

Core Alert

Sometimes, the only way to make a Swing component do exactly what you want it to do is to make use of an interface that is look-and-feel specific. You'll see several examples where this is the case in this book. Strictly speaking, JavaSoft has not finally frozen the interface between the Swing components and their look-and-feel implementations, so if you plan on using this interface, be prepared to evolve your software as you migrate to later releases of Swing. It is likely that these interfaces will stabilize when the final release of JDK 1.2 appears.



`com.sun.java.swing.plaf.multi`

By default, a component only has a single user interface associated with it at any time. However, by using the *multiplexing* look-and-feel provided by the `com.sun.java.swing.plaf.multi` package, it is possible for more than one user interface from any of the other look-and-feel packages, or from a custom pack-

age, to be connected to a single component at one time. This can be useful if, for example, you want a text component to be able to draw its content on the screen and also to be able to “read” that content through a sound card. This is particularly important in an application that uses the Java Accessibility features to make itself more usable by those with sight impairments, for example.

The implementation of the look-and-feel in this package is such that this requirement can be satisfied without any other look-and-feel package knowing that it is being used in connection with another one to manage a single component.

The Model-View-Controller Architecture

The major difference between the Swing components and their AWT counterparts is that the Swing controls are written entirely in Java and, as a result, do not depend on any code provided by the host windowing system to provide their visual appearance or their functionality. On its own, this change makes it possible to create controls that look the same on any platform. However, the controls have not simply been re-implemented in Java—they have, in fact, been completely redesigned using a paradigm that is well-known in object-oriented programming, called the model-view-controller architecture, or MVC for short.

To avoid getting lost in obscure and abstract discussions, let’s look at what the MVC architecture means in terms of a concrete example and then show why this particular way of implementing components is so useful. At the end of the chapter, when you’ve seen what has been done and why it has been done, you’ll find some examples of the results, and you’ll be able to decide for yourself whether or not it was actually worthwhile! If, after this, you’re sold on the idea of having a customized appearance to your applications, or even to somebody else’s applications, in Chapter 13 you’ll see in more detail how to go about implementing your own look-and-feel.

An MVC Component: a Button

To see what the MVC architecture is and how it relates to the Swing components, let’s look at how you might go about designing a component that represents a button. A button is a control that has pieces that represent all three parts of the MVC architecture and it is also simple and well-understood, so it should be easy to use it to examine and assess a new component architecture.

Fundamentally, a button is an object that does little more than sit on a user interface and wait for you to click it. When it has been clicked, it changes its appearance so that it looks pushed-in, generates an event for some interested party to catch, then redraws itself to look popped-out again. It couldn’t be much simpler in principle, but there is a little more to it than this straightforward description would suggest. Before going any further, read back over the first part of this paragraph. By and large, what I’ve told you is what the button looks like while it is being used: it starts in a popped-up state, it

changes appearance so that it appears to be pushed in, and then it pops back out again. These few words actually describe what the *view* part of a Swing component is supposed to know about—how the component should look at any given time.

Core Note

Here and throughout the book you'll find descriptions of components and how they behave. Swing poses a particularly difficult problem for authors in that there is no single way for a component to look or behave—exactly what it looks like or what it does can depend crucially on the look-and-feel that is being used. In this book, unless indicated otherwise, the descriptions match the way components look and behave when the Metal look-and-feel is selected.



In this case, you can see that the button has two different representations that the view has to be able to reproduce. In fact, though, there are more than that. Buttons are not always in an active state: if you create a form with several fields that need to be filled in, with an OK button to be pressed when the form is complete, it makes little sense to offer the possibility for the user to be able to actually press it until all mandatory fields have been completed. To implement this kind of functionality, the button (along with other components) can be either enabled or disabled at any given time. Naturally, the button will only respond to a click when it is enabled. Having a software switch that records the state of the button is useful for the programmer, but of no use to the user, who can't see it, so to make it clear when the button can and can't be pressed, it is normal for the view to render it differently. Thus, there are three different ways to draw a button (only three, because a pressed-in disabled button should not be possible!). More could be added to this, but for now it should be clear that the view has to render the button differently depending on its state.

The state of the button is, of course, something that would be part of any component that represented it. In fact, the button's total state is the *model* part of the MVC architecture. So far, you've seen two items that are legitimate parts of the model—whether the button is pressed in and whether it is enabled. If you started looking more closely at the problem, you would soon see that there are a few more attributes that you would need to have; those attributes would all be held in the model.

You now know that the button holds its state in the model and that the view uses the model to decide how to draw the button. The other important feature of the button is that its state can change—when you click on it with the mouse, or give it the focus and press the space or return key, the button is activated. Obviously, something must be monitoring the mouse and the keyboard and the fact that the button has received or lost the focus, in order to notice that a state change is necessary. The part of the component that receives and responds to input is the *controller*.

Let's represent all of this with a diagram. Figure 1-6 shows a representation of the various pieces of the button in the MVC architecture and it also shows how the state of the various pieces can change. When the button is created, all three pieces come

into existence and get connected together; you'll see later how this is done. The model adopts an initial state: Usually, the button isn't pressed and in this case it will start life enabled. When the button first becomes visible, the view uses the model's initial state to draw it in the appropriate way.

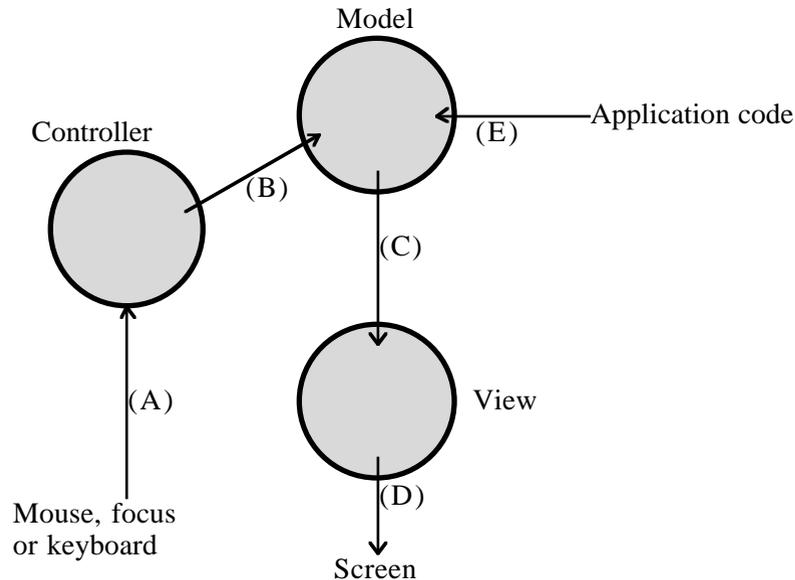


Figure 1-6 A button as an MVC component.

Now suppose that the user clicks the button with the mouse. This action is detected by the controller, which interprets it as a request to click the button. A click actually requires two steps—first, the button is pressed and then it is released. When the mouse button is pressed, the controller tells the model to change its state to reflect the fact that the button itself has been pressed. The button now needs to be redrawn so that it looks pressed in. To make this happen, the model notifies the view that its state has changed, by generating an *event* that the view has registered to receive. On receipt of this event, the view queries the model for its new state and redraws the button accordingly.

When the user releases the mouse button, the controller detects it and changes the model again so that the button's state indicates that it is not pressed. This causes the model to generate another event to the view, as a result of which the button will be redrawn in its *up* state. This particular state change, from *pressed* to *not pressed* also causes the model to generate another event that can be delivered to application code to indicate that the button has been clicked. This is the only piece of this interaction that is important to code outside the button: The rest has only been visible inside the button.

Now suppose the button were disabled instead of enabled. What difference would this have made? When the mouse is pressed over the button, the controller will still detect it and will attempt to change the model's state to reflect this. However, the fact that the button is disabled is held within the model. When the model is disabled it doesn't allow the button's pressed state to be changed, so no event will be generated to the view and the button's appearance won't change. As far as the user is concerned, the button press was ignored.

There is another way in which the button's state can change: Code in the application can change the enabled state of the button, or it can programmatically perform a *click* as if the mouse had been pressed and then released. Disabling the button is an action performed by the model and, of course, causes an event to the view, which will make the button be redrawn with the usual "grayed-out" appearance. Similarly, clicking the button is performed as a two-step interaction with the model in which a mouse press and a mouse release are simulated, without going through the controller. The model, of course, doesn't know which piece of software is changing its state—be it controller or application code, it still sends events to the view and, if necessary, generates the event for the button click.

The MVC Button Implementation in Swing

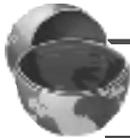
What has just been described is the purist approach to the MVC architecture in which there are three pieces of software that separately implement the model, the view and the controller and which get bound together when the button is created. You have also seen that application code can directly change the model to click the button or change its enabled state.

In practice, while this exact architecture is used for some components (for example the text components that you'll see in Chapter 9), many of the Swing controls are built slightly differently. In these cases, the view and the controller are merged into one entity that implements the functions of both. Also, applications usually don't have to interact directly with the model. Instead, the component itself often (but not always) exposes methods that manipulate the model, so that the almost always deals only with the component.

What do we mean by 'the component' here? Isn't the whole thing the component? In fact, the component is usually made up of several pieces; in the case of the button, these are as follows:

1. An instance of a class that implements the model. In the case of an ordinary button, this class is called `DefaultButtonModel`.
2. An instance of a class that knows how to draw the button that fulfills the role of the view. A class called `BasicButtonUI` does this job for ordinary buttons. As you'll see in Chapter 13, the button's border is provided by yet another class, which we consider to be part of the view for the purposes of this discussion.

3. An instance of a class that responds to user input, in the role of controller. For the button, this role is played by the `BasicButtonListener` class.
4. A wrapper class that provides the programming interface to the button and hides the other pieces. The Swing class that represents a button is `JButton`.



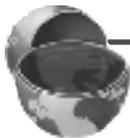
Core Note

You can see that the button actually has separate view and controller classes. Many other components do not.

In this case, when we speak of the “the component,” we would be referring to the `JButton` instance, because this is what the application program would almost always interact with. However, all of the Swing components that have a model provide a method that allows applications to access it directly if they need to. For some controls, there is no way to change the model via the component—you have to go directly to the model.

Using MVC to Enable the Pluggable Look-and-Feel Architecture

The MVC architecture is the cornerstone of the pluggable look-and-feel feature in Swing. If you wanted to take a button and make it look different and, perhaps, make it respond to different keys or mouse events, you would need to modify the controller and the view. You wouldn’t change the model, because you need the same button states—after all, this is still a button, no matter how you draw it and you wouldn’t want to change the programming interface to the wrapper component either because if you did, you would make it difficult to substitute your button for an existing one. So, if you wanted to be able to substitute one button look-and-feel for another easily, you would need to design your components in such a way that the view and controller could be removed and replaced by another one without disturbing anything else. This, of course, is one reason why the view and controller are often implemented as a single unit.



Core Note

You can change the button's model for another one if you want to, provided that it implements the same interface as the default model. This is, however, an application issue and is not strictly part of the pluggable look-and-feel architecture because it does not affect the button's appearance.

Plugging in new view-controller units means having a well-defined interface between this part and the component itself and having an interface that tells the component that the look-and-feel specific part is being changed. In fact, the actual interface that is needed will depend on the type of component because, as you can imagine, a tree control is vastly more complex than a button. There is, however, a core set of methods that is required by every pluggable unit. This core set is specified in Swing by the abstract class `com.sun.java.swing.plaf.ComponentUI`. The suffix `UI` here is common to all of the classes in `com.sun.java.swing.plaf` that specify an interface between a particular component and its view-controller implementation. Other examples are `TreeUI`, `TableUI`, `LabelUI` and so on. Each of these is an abstract class that is derived from `ComponentUI`.

Core Note

In cases where the view and controller are separate, the view class, naturally enough, is the one that has the `UI` suffix.



If you wanted to change the way a Swing component looks or behaves, you would write a class that implemented all the methods in the corresponding `UI` class for that type of component and arrange for it to be plugged in at run time. For example, to create a new look-and-feel for a tree, you would have to implement a class that provided all the methods of `TreeUI`. The Swing packages, as we know, contain several look-and-feel packages. These packages all consist of a set of classes that implement (or inherit) all of the methods of all of the `UI` classes in the `com.sun.java.swing.plaf` directory.

The `ComponentUI` class has only a very small number of methods; every pluggable component module must, of course, provide all of them:

```
public abstract class ComponentUI() {
    public static ComponentUI createUI(JComponent c);
    public void installUI(JComponent c);
    public void uninstallUI(JComponent c);
    public void update(Graphics g, JComponent c);
    public void paint(Graphics g, JComponent c);
    public Dimension getPreferredSize(JComponent c);
    public Dimension getMinimumSize(JComponent c);
    public Dimension getMaximumSize(JComponent c);
    public boolean contains(JComponent c, int x, int y);
}
```

The gory details of how the actual pluggable look-and-feel mechanism works are not discussed in this chapter, because most of the time, you don't really need to know how the component is created in order to use it. These details are important only if

you want to create a replacement look-and-feel, so we'll confine ourselves to a basic outline here. If you want to know more, you'll find complete coverage in Chapter 13.

The first three methods deal with creating a UI object for a control and connecting it to the component. When, for example, a button is created, the `createUI` method of the button UI class for the look-and-feel that happens to be active is called. This is a static method and its job is to return an object that can be plugged into a `JButton`. If the Metal look-and-feel is selected, the `createUI` method of `MetalButtonUI` will be called and it will return a new `MetalButtonUI`, which will be installed into the `JButton`. To tell the new `MetalButtonUI` object that this has happened and to let it know which component it is associated with, its `installUI` method is called. As you can see, this method receives a reference to the component into which the UI class is being installed as an argument.

It is possible to replace the UI after the component has been created. If this is happening, the old UI class is disconnected from the component by calling its `uninstallUI` method, then the `installUI` method of the replacement UI is invoked. The job of creating UI classes for components, based on the currently selected look-and-feel, is carried out by a class called `UIManager` that resides in the `com.sun.java.swing` package.

The remaining methods are invoked when the component is in use and they deal with things that only the look-and-feel class can know about. Clearly, drawing the component is very look-and-feel specific, so most (but not all) of this is handled by the UI class's `update` and `paint` methods. Similarly, a component's size can depend on how it is being drawn, so the `JComponent` methods that determine how big a component can be or wants to be do their job by calling the corresponding method in the UI class. Finally, the `contains` method, as its name suggests, determines whether a given point is "inside" the component; if the drawn shape of the component is irregular, deciding whether a point is inside or not cannot be done by generic code—instead, the question is passed to the UI class, which knows how the component is drawn.

Selecting the Look-and-Feel

When an application starts, its `UIManager` needs to know which of the available look-and-feel classes to use. The user can determine the selected UI by editing the file `swing.properties`. This file resides in the `lib` directory of the user's Java installation so that, for example, if the Java software has been installed in `C:\java`, this file will be called `C:\java\lib\swing.properties`.

The content of this file is a set of `property = value` lines, which will be described in Chapter 13. The only line of interest here is the following one:

```
swing.defaultlaf = com.sun.java.swing.plaf.metal.MetalLookAndFeel
```

The `swing.defaultlaf` property must be set to the name of the class that provides the look-and-feel support. This example selects the Metal look-and-feel. To select Motif, use this line:

```
swing.defaultlaf = com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

and to use the Windows look-and-feel, you need this line:

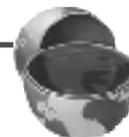
```
swing.defaultlaf = com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

If `swing.properties` doesn't exist, or it doesn't contain a `swing.defaultlaf` property, the cross-platform Metal look-and-feel will be used.

Core Note

If you are using JDK 1.2, these lines will, of course, need to be changed to look like this:

```
swing.defaultlaf = java.awt.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf = java.awt.swing.plaf.motif.MotifLookAndFeel
swing.defaultlaf = java.awt.swing.plaf.windows.WindowsLookAndFeel
```



When you are running the examples in this book or developing your own programs, you'll find it useful to switch between the various look-and-feel classes to see how a program looks with the different styles installed. Chapter 13 shows you how to give an application the ability to switch its look-and-feel on command, but, if you aren't using a program that can do that, the only way around it is to stop the program, edit `swing.properties` and then start the application again. To save time, I keep all three of the above lines in my file and comment out the two that I don't want by placing a “#” at the start of the line. Here's what my file looks like when I want to run an example with the Motif look-and-feel:

```
#swing.defaultlaf = com.sun.java.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf = com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf = com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

One Application, Three Disguises

To conclude this chapter, let's see how a typical Swing application's appearance changes when its look-and-feel is switched, by looking ahead to a program that will be developed in Chapter 10. This program demonstrates the Swing tree component by using it to display a view of a file system on a PC. The meaning of what you're actually seeing here is not really important—what is important is the way in which the tree's content is presented. Look carefully at Figures 1-7, 1-8, and 1-9 and notice the differences in the way that the trees are drawn. As you look at these figures,

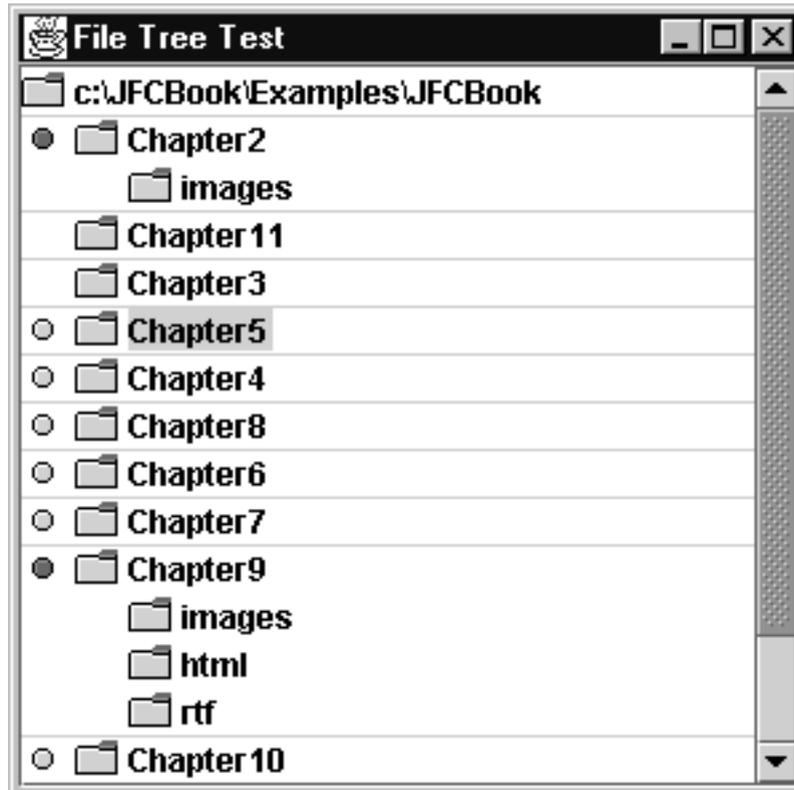


Figure 1-7 The Metal look-and-feel.

remember that the program was only written once and no code was changed between these three screen shots.

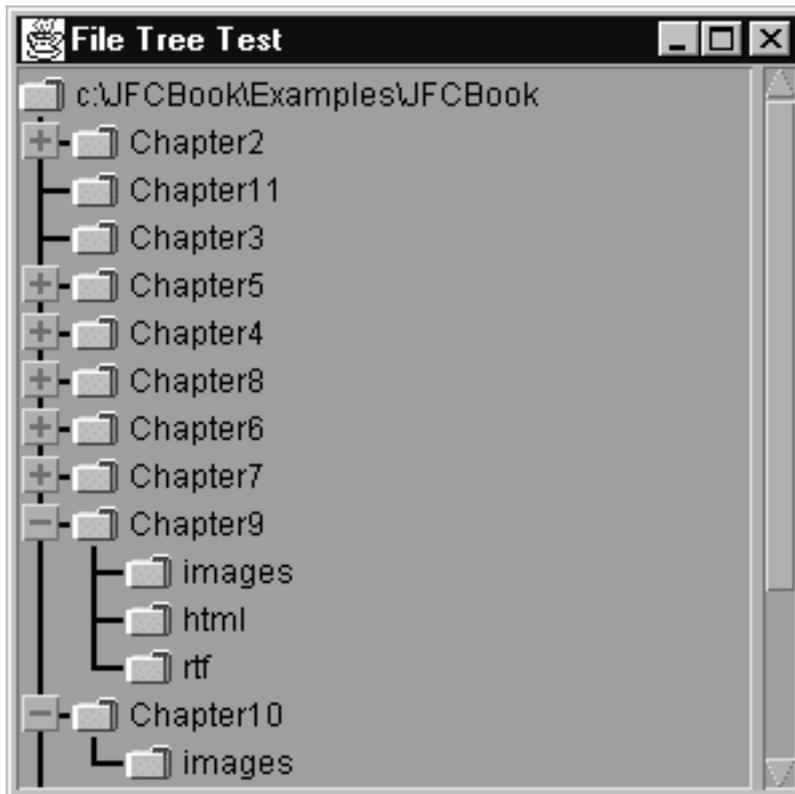


Figure 1-8 The Motif look-and-feel.

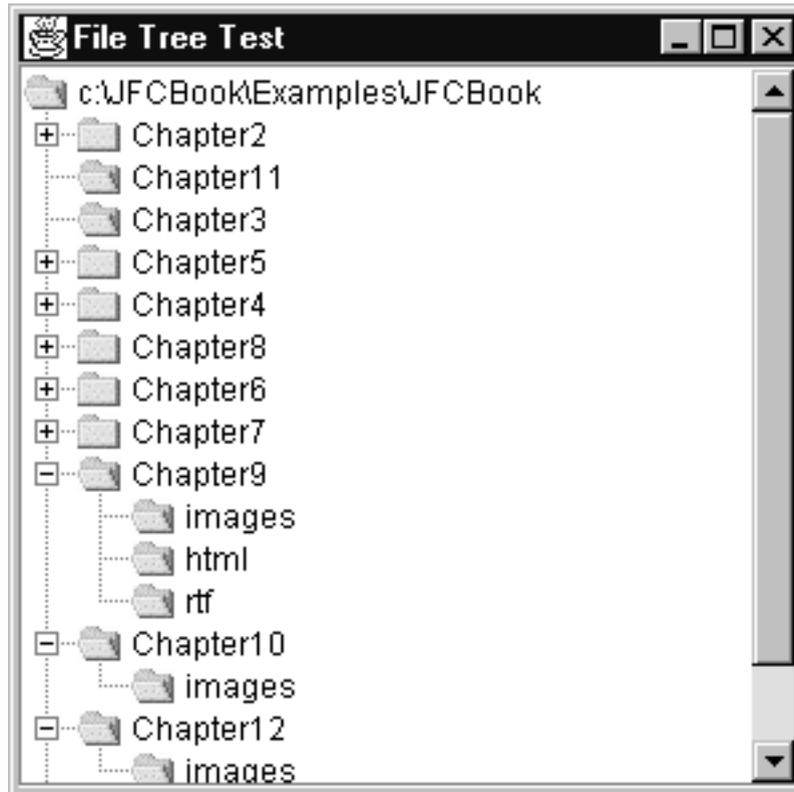


Figure 1-9 The Windows look-and-feel.

Summary

This chapter introduced the Java Foundation Classes and, in particular, the Swing component set, which is the major new feature of JFC 1.1. It presented the history of the JFC and described how the Swing components relate to their AWT predecessors.

You also saw the major features of the Swing set and were introduced to the “core” class, `JComponent`, which makes most of them possible. After a brief description of the Swing packages, you were shown the architecture of the Swing components and how this architecture makes it possible to change the appearance of a component or of a complete application without changing any application code.

In the rest of this book, you’ll be introduced to all of the Swing components and to many of the new facilities that Swing provides. The next chapter, however, covers some basic ground by discussing the fundamentals of the AWT, and then introduces two of the simpler Swing controls—`JLabel` and `JButton`.