CHAPTER 5

SWING

The Origins of Swing

- Swing did not exist in the early days of Java.
- Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
- One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers.
- This means that the look and feel of a component is defined by the platform, not by Java.
- Because the AWT components use native code resources, they are referred to as heavyweight.

- The use of native peers led to several problems.
- First, because of variations between operating systems, a component might look, or even act, differently on different platforms.
- This potential variability threatened the overarching philosophy of Java: write once, run anywhere.
- Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
- Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque

- Not long after Java's original release, it became apparent that the limitations andrestrictions present in the AWT were sufficiently serious that a better approach was needed.
- The solution was Swing.
- Introduced in 1997, Swing was included as part of the JavaFoundation Classes (JFC).
- Swing was initially available for use with Java 1.1 as a separate library.
- However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java

Swing Is Built on the AWT

- Although Swing eliminates a number of the limitations inherent in the AWT, Swing does not replace it.
- Instead, Swing is built on the foundation of the AWT.
- This is why the AWT is still a crucial part of Java.
- Swing also uses the same event handling mechanism as the AWT.
- Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

Two Key Swing Features

- As just explained, Swing was created to address the limitations present in the AWT.
- It does this through two key features: lightweight components and a pluggable look and feel.
- Together they provide an elegant, yet easy-to-use solution to the problems of the AWT.
- More than anything else, it is these two features that define the essence of Swing

1. Swing Components Are Lightweight

- With very few exceptions, Swing components are lightweight.
- This means that they are written entirely in Java and do not map directly to platform-specific peers.
- Because light weight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes.
- Thus, lightweight components are more efficient and more flexible.
- Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- This means that each component will work in a consistent manner across all platforms.

2. Swing Supports a Pluggable Look and Feel

- Swing supports a pluggable look and feel (PLAF).
- Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing.
- This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects.
- In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component.

- Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles.
- To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style
- Pluggable look-and-feels offer several important advantages.
- It is possible to define a look and feel that is consistent across all platforms.
- Conversely, it is possible to create a look and feel that acts like a specific platform.
- For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
- It is also possible to design a custom look and feel.

- Finally, the look and feel can be changed dynamically at run time.
- Java SE 6 provides look-and-feels, such as metal and Motif, that are available to all Swing users.
- The metal look and feel is also called the Java look and feel.
- It is platform-independent and available in all Java execution environments.
- It is also the default look and feel.
- Windows environments also have access to the Windows and Windows Classic look and feel.
- This topic uses the default Java look and feel (metal) because it is platform independent

The MVC Connection

- In general, a visual component is a composite of three distinct aspects:·
  - The way that the component looks when rendered on the screen
  - The way that the component reacts to the user
  - The state information associated with the component
- No matter what architecture is used to implement a component, it must implicitly contain these three parts.
- Over the years, one component architecture has proven itself to be exceptionally effective:
- Model-View-Controller, or MVC for short.

- The MVC architecture is successful because each piece of the design corresponds to an aspect of a component.
- In MVC terminology, the model corresponds to the state information associated with the component.
- For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.
- The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.
- The controller determines how the component reacts to the user.

- For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user 's choice (checked or unchecked).
- This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two.
- For instance, different view implementations can render the same component in different ways without affecting the model or the controller
- Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components.

- Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate.
- For this reason, Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture.
- Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.
- Swing's pluggable look and feel is made possible by its Model-Delegate architecture.
- Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program.
- Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input

- To support the Model-Delegate architecture, most Swing components contain two objects.
  - The first represents the model.
  - The second represents the UI delegate.
- Models are defined by interfaces.
- For example, the model for a button is defined by the ButtonModel interface.
- UI delegates are classes that inherit ComponentUI.
- For example, the UI delegate for a button is ButtonUI.
- Normally, your programs will not interact directly with the UI delegate.

Components and Containers

- A Swing GUI consists of two key items: components and containers. However, this distinction is mostly conceptual because all containers are also components.
- The difference between the two is found in their intended purpose: As the term is commonly used, a component is an independent visual control, such as a push button or slider.
- A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container.
- Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers.
- This enables Swing to define what is called a containment hierarchy, at the top of which must be a top-level container