Swing from A to Z

Getting Started, Part 1

by Richard G. Baldwin
baldwin.richard@iname.com

Java Programming, Lecture Notes # 1000

July 17, 2000

Preface

Not the first Swing lesson

Even though I have entitled this lesson on Swing "Getting Started", this is not the first lesson that I have written on Swing. Rather, I have previously written several other lessons, which you will find at *Baldwin's Java Programming Tutorials*.

Previous lessons

The previous lessons include such titles as:

- The AWT and Swing, A Preview
- Swing and the Delegation Event Model
- Swing, New Event Types in Swing
- Swing, Understanding **getContentPane()** and other JFrame Layers
- The Swing Package, A Preview of Pluggable Look and Feel
- Swing, Hidden Buttons with Icons, Icon Images, Borders, Tool Tips, Nested Buttons, and Other Fun Stuff
- Swing, Creating and Using Trees
- Swing, Custom Rendering of Tree Nodes
- Swing, Simplified Lists in Swing
- Swing, Understanding Component MVC Models
- Swing, Custom Rendering of JList Cells
- Swing, Custom List Selection Model for JList Objects

Why "Getting Started?"

So, having already written and published a dozen lessons on Swing, why did I entitle this lesson "Getting Started?" Because in many cases, the earlier lessons show you how to do certain things using Swing, but don't provide an understanding of why things work the way that they do.

A fresh start

In this series of lessons, I plan to get down to fundamentals and provide explanations of many operations that were not clearly explained in the previous lessons.

Perhaps a better name for this lesson would be "A Fresh Start."

Introduction

Previous lessons are very important

The lessons that I have previously written on Swing contain a lot of information that I won't repeat in this series of lessons. Therefore, I recommend that you study them in conjunction with this series of lessons.

JavaBean Components

In addition, I will make numerous references to JavaBean Components in this series of lessons. I have previously written several lessons on JavaBeans. I will also provide a brief description of JavaBean Components in Part 2 of this lesson.

Lightweight components

I will make numerous references to *lightweight components* in this series of lessons, so I recommend that you study my previous lessons on that topic as well. I will also briefly discuss lightweight components later in this lesson.

Model view control paradigm

In order to truly understand Swing, you need to understand the Model View Control (MVC) paradigm. Guess what? I have already written some lessons on MVC. For those in a hurry, I will provide a brief discussion of MVC in Part 2 of this lesson.

Delegation event model

Finally, in order to understand Swing, you must understand the Delegation Event Model, which I will discuss briefly in the next section. As you may already have guessed, I have also written numerous lessons on the Delegation Event Model.

You will find links to the lessons on the Delegation Event Model and all of the other lessons mentioned above at my web site.

The Swing software

Although a version of Swing existed prior to the release of JDK 1.2, it was not fully integrated into the JDK. When JDK 1.2 was released, it included Swing as an integral part of the JDK. Therefore, the release of the JDK 1.2 version was a major milestone in the evolution of Swing.

If you have downloaded and installed JDK 1.2.x or a later version of the JDK, you already have Swing installed and can begin writing and executing programs using it.

I recommend that you use the JDK 1.2 version of Swing, or later versions as they become available. (*At the time of this writing, JDK 1.3 is available at http://java.sun.com/j2se/1.3/*)

Earlier versions of Swing

If you need to use a version of Swing that existed prior to JDK 1.2, you have some special download and installation tasks to perform, and you must adhere to some special programming syntax.

In that case, you should visit some of my early lessons on Swing, which were written using an early version of Swing.

Will use JDK 1.2x or later

In this series of lessons, I will be using JDK 1.2x or later versions of Swing.

Event Handling

Swing depends on the Delegation Event Model introduced in JDK 1.1. As mentioned above, I have written numerous lessons on the use of this event model, which you will find in my earlier Tutorials.

Short cuts

In some cases, Swing includes automatic shortcuts that disguise the fact that the Delegation model is being used, but even in those cases, it is being used behind the scenes.

Critical to understand the event model

It is absolutely critical that you understand how the Delegation Event Model works in order to understand Swing. While I strongly recommend a serious study of the topic, for those of you in a hurry, a brief description of the Delegation Event Model follows.

You must also understand the Java interface

I tell my students at least once each week that if they don't understand the Java *interface*, they can't possibly understand Java. So, if you don't understand the *interface*, you have another task ahead of you. You will find lessons on the Java interface on my web site.

(Again, for those in a hurry, I also provide a brief description of the interface later in this lesson.)

A *callback* system

I like to think of the Delegation event model as a *callback* system. The event model consists of event *sources* and event *listeners*.

What is a source?

Event sources are program elements capable of

- Creating and maintaining a list of interested listeners,
- Detecting that an interesting event has happened (such as the price of a particular stock crossing a specified threshold), and
- Notifying interested listeners that the event has happened, passing some information about the event to the listener with the notification.

What is a listener?

A listener is a program element that has the ability to register itself on a source. Registering itself on a source means that it is requesting to be notified *(called back)* when the event occurs.

Implement some behavior

The purpose of being notified or called back, of course, is to make it possible for the listener to take some specific action or implement some behavior as a result of the event.

An example action might be to purchase or sell shares of the stock whose price just crossed the threshold.

Multicasting events

The term commonly used for the notification of listener objects by a source is *multicasting*.

Now, about the *interface* ...

The Java *callback* system depends on the use of the Java *interface*. All listener objects must *implement* one or more listener *interfaces* in order to be eligible for registration with one or more sources.

The source is very particular

This is because the source is willing to register only listener objects of a specific type for notification of a specific type of event. The type of listener is determined by the interface that the listener object's class implements. *(It is a central feature of the Java interface, that the type of an object is determined not only by the class from which the object is instantiated, but also by the interface that the object's class implements.)*

Event types and listener interfaces come in pairs

There is a direct correspondence between the type of the event and the interface that the listener object's class implements. In fact, the standard library contains many matched pairs of event types and listener interfaces (*MouseEvent* class and *MouseListener* interface, for example).

Sources can register many different listeners

Typically a source can register many different listener objects for the same type of event. When the event happens, all registered listeners will be notified.

Register for different types of events

Typically, a source can also register listeners for several different types of events.

For example, a **Button** object can register many different listener objects to be notified of *mouse* events that occur on the button and can register many different listener objects to be notified of *action* events that occur on the same button. In other words, a **Button** object can multicast both *mouse* events and *action* events (and possibly other types of events as well).

Listener can register on multiple sources

Typically, a listener object of a particular type can be registered to be notified of events that occur on many different sources.

For example, a single *action* listener object can be registered to be notified of *action* events occurring on a **Button**, a **TextField**, and a **MenuItem**.

A big point-to-point wiring system

Therefore, you can think of a Java *event-driven* program as consisting potentially of many different sources and many different listeners wired together in many different ways.

Listener objects are registered on specific sources to be notified when specific events occur. Each listener object can be registered on multiple sources.

Event sources know when such events occur, and notify all registered listeners when they occur.

Lightweight Components

Java makes a distinction between *heavyweight* and *lightweight* components.

What is a heavyweight component?

To put it simply, Java depends on the underlying operating system to render all heavyweight components. *(To render a component is to display its image on the screen.)*

This has several disadvantages, the most obvious of which is that the components look different when the same Java program is run under different operating systems. Beyond that, I'm not going to discuss the disadvantages here. Please see my earlier lessons on this topic for more information.

What is a lightweight component?

Lightweight components are components that are rendered exclusively using the drawing primitives of Java.

Look and feel is independent of OS

As a result, to a very large extent, the look and feel of a *lightweight* component is independent of the operating system under which the program is running.

Advantages

This (in conjunction with MVC) provides a number of advantages, including:

- Lightweight components look the same regardless of the operating system under which the program is running.
- The look and feel of a lightweight component can be changed at runtime.
- Lightweight components can have transparent backgrounds.

Swing components are lightweight

All Swing components are *lightweight* components except for the following top-level containers: **JApplet**, **JFrame**, **JDialog**, and **JWindow**.

What is a container?

A container is a component that can contain other components in a parent-child sense.

What is a top-level container?

A top-level container is a container that can be rendered on the computer screen *(operating system desktop)* alongside other applications that are running (such as Microsoft Word or Netscape Navigator).

Most Swing components must be contained

Except for the top-level containers, all Java components must be contained by a top-level container, or by one of the containers that it contains.

What about a Swing button?

A Swing button is a container, but it is not a top-level container.

Therefore, you cannot cause a Swing button to be rendered directly on the computer screen alongside Microsoft Word or some other application that is running.

However, you can cause a Swing button to be rendered as a child of a **JFrame** object, which can be rendered on the operating system desktop alongside Microsoft Word or Netscape Navigator.

Mixing heavyweight and lightweight components

It is possible for you to mix heavyweight and lightweight components, although I would advise you to do so only if absolutely necessary. If you do, be prepared for some possible unusual behavior, including the fact that heavyweight components will always obscure lightweight components if they overlap.

For more information on lightweight components, see my previous *Tutorials*.

Where Do We Go From Here?

In Part 2 of this lesson, I will discuss the MVC paradigm as it relates to Swing along with the Delegation Event Model.

About the author

Richard Baldwin *is a college professor and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two. He frequently provides onsite Java and/or XML training at the high-

tech companies located in and around Austin, Texas. He is the author of Baldwin's Java Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin.richard@iname.com

-end-