Swing from A to Z

Getting Started, Part 2

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

Java Programming, Lecture Notes # 1001

July 24, 2000

Introduction

This is the second part of a two-part lesson in which I am discussing background information necessary for an understanding of Swing.

If you missed the first part, you should go back and read it before continuing with this part.

MVC, JavaBean Components, etc.

In Part 1 of this lesson, I discussed Event Handling and Lightweight components. In this lesson, I will discuss MVC, JavaBean Components, and the Java interface.

Model view control paradigm

In order to really understand Swing, you need to understand the Model View Control (MVC) paradigm. I have previously published lessons on MVC that will help you to come up to speed on this topic. For those in a hurry, I will provide a brief discussion of MVC later in this lesson.

JavaBean Components

In this series of lessons, I will make numerous references to JavaBean Components. I have previously published several lessons on JavaBean components.

I will also provide a brief description of JavaBean Components later in this lesson

The Java interface

The Java interface is the backbone of Java. As I will discuss later, in addition to its many other uses, an understanding of the interface is critical to the use of the Delegation Event Model with Swing. You may also want to study my previous lessons on the Java interface and the Delegation Event Model as well.

Model View Control (MVC)

What is MVC?

MVC is a programming structure in which the program is separated into three distinct parts.

One part maintains the data that describes the state of the program. This part is often called the *model*.

A second part provides the conduit by which data enters the program. This part is often called the *control*.

A third part provides one or more views of the data. As indicated by the name, this part is often called the *view*.

Observable and observer

In the previous lesson, I discussed the use of the Java interface to implements *callbacks*.

Java provides a class/interface combination (*observable/observer*) that makes it possible for views to register themselves on a model to be notified whenever the data stored in the model changes. You can read about this in one of the previous lessons in my *Tutorials*.

Swing components adhere to MVC concepts

Most Swing components are designed according to *(a somewhat modified version of)* the MVC paradigm.

This has a number of advantages. Perhaps the most obvious advantage is that the rendering (view) of a particular component can be changed at runtime.

Pluggable look and feel

For example, with Swing's *Pluggable Look and Feel*, it is possible to cause a swing button to look like a *Motif* button, a *Metal* button, or a *Windows* button, and to change that look and feel at runtime if desired. *(Although I have never attempted to install and use it, I understand that a Macintosh look and feel is available also.)*

What is Metal?

Metal is a proprietary look and feel that Sun designed specifically for use with Java.

What about Motif, Windows, and Macintosh?

Motif is a common "look and feel" from Sun, while *Windows* is from Microsoft and *Macintosh* is from Apple.

You can roll your own

It is also possible for you to create your own look and feel if you have such a need.

Not pure MVC

Swing doesn't use a *pure* MVC paradigm for components. Rather, in Swing, the control and the view are combined, producing a paradigm often referred to as the *model-delegate*.

Changing the view at runtime

If your program changes the look and feel at runtime, the data in the model isn't changed. Only the view of that data is changed. The component continues to represent the same data to the user, but presents it in a different way.

MVC is important to Swing

In this series of lessons, we will see various situations where the separation of the component into a model and a delegate provides important capabilities.

JavaBean Components

JavaBean Components are Java program elements that adhere to a well-defined set of interface specifications. Adherence to these specifications causes beans to be highly-reusable.

See previous lessons on beans

You can learn more about JavaBean Components in several previous lessons in my *Tutorials*.

Swing components are beans

Swing components meet the JavaBean specifications. Therefore, Swing components are JavaBean components, often referred to simply as *beans*.

Bean properties

Beans maintain state information in the form of *properties*.

Another callback mechanism

One of the important capabilities of a bean is the ability to register interested listeners (observers) and to notify them when the value of a property changes.

Bound properties

This can happen at two levels. At one level, the listener is simply notified of a change in the value of a property. Properties of this sort are referred to as *bound* properties.

Constrained properties

At another level, the listener is not only notified, but has the right to veto the change. Properties of this sort are referred to as *constrained* properties.

Need to understand the inner workings of bound and constrained properties

Because many Swing components have bound and constrained properties, an understanding of bound and constrained properties is important to an understanding of Swing.

You can learn about bound and constrained bean properties in the lessons on my web site.

Swing components have the ability to register and notify other objects when the value of a property changes. Often, this notification happens automatically, but you still need to understand what is happening.

The Java Interface

A Java *interface* declares a set of (optional) *method signatures* (no bodies) and a set of (optional) *constants* (public final variables).

Implementing an interface

A Java class can *implement* an interface.

If a Java class implements an interface, it must provide a *concrete definition* of all methods declared in the interface. (An empty method with a matching signature in the class definition will satisfy this requirement.)

A class that implements the interface also has access to all of the constants declared in the interface.

Object references are typed

All references to a Java object must adhere to a *type*. The *true type* of a Java object is the name of the class from which it is instantiated. This is the type specification that provides access to all of the members of an object instantiated from the class (assuming that access to the members is not otherwise blocked.).

Thus, a reference to a Java object can be treated as the type of the class from which the object was instantiated.

Other possibilities exist also

The reference to the object can also be treated as the type of any superclass of the class from which it was instantiated. However, when referring to an object by its superclass type, some members may not be accessible. *(Downcasting may be required to access some members.)*

Treating object reference as type Object

The superclass of all classes is the class named **Object**. It is common practice to treat references to Java objects as type **Object**, particularly in the design of data structures and containers.

This makes it possible to design a totally generic container and use it to store references to objects instantiated from any class.

Downcasting is probably necessary

However, when those references are later used in an attempt to access the members of the object, it is typically necessary to *downcast* the reference to the true type of the object.

The interface type

A reference to a Java object can also be treated as a type that is the name of any interface implemented by the class from which the object was instantiated, or any interface implemented by any superclass of that class. *(Any members of the object not declared in the interface will not be accessible in this case.)*

Interface is critical to registration and callback

The Java interface has many important uses. For example, it is the backbone of the Java registration-callback system.

Under the Delegation Event Model discussed in the previous lesson, a source for a particular event type is willing to register listener objects for notification only if they are instantiated from a class that implements a specific interface.

Another example

In standard Java MVC, a Java *model* must be an object of a class that extends the **Observable** class.

Objects instantiated from such a class inherit the ability to *register* objects instantiated from classes that implement the **Observer** interface *(view objects)*. Any object that doesn't implement the **Observer** interface cannot be registered on an **Observable** object *(a model)*.

Advantages

One of the advantages of such a scheme is that the author of the model class doesn't have to know the names of the classes from which the view objects are instantiated. It is sufficient to know that they are instantiated from classes that implement the **Observer** interface.

Another advantage is that the author of the model class knows that all registered observers provide concrete definitions of all methods declared in the **Observer** interface.

Interface methods can be safely invoked on observers

The methods declared in the interface can be safely invoked on all registered **Observer** objects with no downcasting required.

No guarantee of correct behavior

Of course, there is no guarantee that the concrete definition of the interface method will behave properly. The guarantee is simply that the method exists and can be invoked.

Where Do We Go From Here?

Most Swing components extend from the class named **javax.swing.JComponent**.

JComponent extends **java.awt.Container** which, in turn extends **java.awt.Component**.

If you understand the capabilities provided by these three classes, you already know a great deal about all Swing components.

One of the beauties of inheritance is that it is not necessary for you to learn about specific capabilities on a component-by-component basis. Rather, you can learn about capabilities on a group basis by learning about the common superclass of those components.

Most Swing components are containers

Unlike the AWT components, almost all Swing components behave as containers. This means that they can contain other things, including other Swing components.

Containment is very beneficial

A lot of the improvement that Swing provides relative to the AWT results from the capability of a Swing component to contain other things, such as images and other components.

A button tree

For example, you could create a Swing tree whose leaves are buttons where each button contains some text and an image.

Each of the buttons could register a variety of different event listeners including **Action** listeners, **Mouse** listeners, and **Focus** listeners.

I'm not suggesting that such a structure would be useful. I mention this possibility simply to illustrate the range of possibilities that exist with Swing.

Will devote a lot of time to JComponent and its superclasses

Since an understanding of the **JComponent**, class and its superclasses is so critical to an understanding of Swing capabilities, the next several lessons will be devoted to an understanding of the capabilities that Swing components inherit from these classes.

Where's the code?

If you have spent much time with my tutorial lessons in the past, you probably know that I rarely write this much text without showing some code to back it up. Beginning in the next lesson, I will get back into that mode and start showing you how to program using Swing.

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-