

JavaFX GUI Programming

JavaFX is Java's next-generation client platform and GUI framework. JavaFX provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs.

Before continuing, it is important to mention that the development of JavaFX occurred in two main phases. The original JavaFX was based on a scripting language called JavaFX Script. However, JavaFX Script has been discontinued. Beginning with the release of JavaFX 2.0, JavaFX has been programmed in Java itself and provides a comprehensive API. JavaFX also supports FXML, which can be (but is not required to be) used to specify the user interface. JavaFX has been bundled with Java since JDK 7, update 4. The latest version of JavaFX is JavaFX 8, which is bundled with JDK 8. (The version number is 8 to align with the JDK version. Thus, the numbers 3 through 7 were skipped.) Because, at the time of this writing, JavaFX 8 represents the latest version of JavaFX, it is the version of JavaFX discussed here. Furthermore, when the term JavaFX is used in this and the following chapters, it refers to JavaFX 8.

JavaFX Basic Concepts

In general, the JavaFX framework has all of the good features of Swing. For example, JavaFX is lightweight. It can also support an MVC architecture. Much of what you already know about creating GUIs using Swing is conceptually applicable to JavaFX. That said, there are significant differences between the two.

From a programmer's point of view, the first differences you notice between JavaFX and Swing are the organization of the framework and the relationship of the main components. Simply put, JavaFX offers a more streamlined, easier-to-use, updated approach. JavaFX also greatly simplifies the rendering of objects because it handles repainting automatically. It is no longer necessary for your program to handle this task manually. The preceding is not intended to imply that Swing is poorly designed. It is not. It is just that the art and science of programming has moved forward, and JavaFX has received the benefits of that evolution. Simply put, JavaFX facilitates a more visually dynamic approach to GUIs.

The JavaFX Packages

The JavaFX elements are contained in packages that begin with the **javafx** prefix. At the time of this writing, there are more than 30 JavaFX packages in its API library. Here are four examples: **javafx.application**, **javafx.stage**, **javafx.scene**, and **javafx.scene.layout**. Although we will only use a few of these packages in this chapter, you will want to spend

some time browsing their capabilities. JavaFX offers a wide array of functionality.

The Stage and Scene Classes

The central metaphor implemented by JavaFX is the *stage*. As in the case of an actual stage play, a stage contains a *scene*. Thus, loosely speaking, a stage defines a space and a scene defines what goes in that space. Or, put another way, a stage is a container for scenes and a scene is a container for the items that comprise the scene. As a result, all JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes. To create a JavaFX application, you will, at minimum, add at least one **Scene** object to a **Stage**. Let's look a bit more closely at these two classes. **Stage** is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

As mentioned, **Scene** is a container for the items that comprise the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, you will add those elements to an instance of **Scene**.

Nodes and Scene Graphs

The individual elements of a scene are called *nodes*. For example, a push button control is a node. However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called leaves. The collection of all nodes in a scene creates what is referred to as a *scene graph*, which comprises a *tree*. There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.

The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent**, **Group**, **Region**, and **Control**, to name a few.

Layouts

JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout. Several other layouts, such as **BorderPane** (which is similar to the AWT's **BorderLayout**), are available. The layout panes are packaged in **javafx.scene.layout**.

The Application Class and the Life-cycle Methods

A JavaFX application must be a subclass of the **Application** class, which is packaged in **javafx.application**. Thus, your application class will extend **Application**. The **Application** class defines three life-cycle methods that your application can override. These are called **init()**, **start()**, and **stop()**, and are shown here, in the order in which they are called:

```
void init( )  
  
abstract void start(Stage primaryStage)  
  
void stop( )
```

The **init()** method is called when the application begins execution. It is used to perform various initializations. As will be explained, however, it *cannot* be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.

The **start()** method is called after **init()**. This is where your application begins and it *can* be used to construct and set the scene. Notice that it is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage. (You can also create other stages, but you won't need to for simple applications.) Notice that this method is abstract. Thus, it must be overridden by your application.

When your application is terminated, the **stop()** method is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.

Launching a JavaFX Application

To start a free-standing JavaFX application, you must call the **launch()** method defined by **Application**. It has two forms. Here is the one used in this chapter:

```
public static void launch(String ... args)
```

Here, *args* is a possibly empty list of strings that typically specify command-line arguments. When called, **launch()** causes the application to be constructed, followed by calls to **init()** and **start()**. The **launch()** method will not return until after the application has terminated. This version of **launch()** starts the subclass of **Application** from which **launch()** is called. The second form of **launch()** lets you specify a class other than the enclosing class to start. Before moving on, it is necessary to make an important point: JavaFX applications that have been packaged by using the **javafxpackager** tool (or its equivalent in an IDE) do not

need to include a call to **launch()**. However, its inclusion often simplifies the test/debug cycle, and it lets the program be used without the creation of a JAR file. Thus, it is included in all of the JavaFX programs in this book.

A JavaFX Application Skeleton

All JavaFX applications share the same basic skeleton. Therefore, before looking at any more JavaFX features, it will be useful to see what that skeleton looks like. In addition to showing the general form of a JavaFX application, the skeleton also illustrates how to launch the application and demonstrates when the life-cycle methods are called. A message noting when each life-cycle method is called is displayed on the console. The complete skeleton is shown here:

JavaFXSkel.java

```
// A JavaFX application skeleton.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

    public static void main(String[] args) {
        System.out.println("Launching JavaFX application.");

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the init() method.
    public void init() {
        System.out.println("Inside the init() method.");
    }

    // Override the start() method.
    public void start(Stage myStage) {
        System.out.println("Inside the start() method.");

        // Give the stage a title.
        myStage.setTitle("JavaFX Skeleton.");
    }
}
```

```
// Create a root node. In this case, a flow layout pane
// is used, but several alternatives exist.
FlowPane rootNode = new FlowPane();

// Create a scene.
Scene myScene = new Scene(rootNode, 300, 200);

// Set the scene on the stage.
myStage.setScene(myScene);

// Show the stage and its scene.
myStage.show();
}

// Override the stop() method.
public void stop() {
    System.out.println("Inside the stop() method.");
}
}
```



Figure 1: Output of JavaFXSek1.java

Compiling and Running a JavaFX Program

One important advantage of JavaFX is that the same program can be run in a variety of different execution environments. For example, you can run a JavaFX program as a stand-alone desktop application, inside a web browser, or as a Web Start application. However,

different ancillary files may be needed in some cases, for example, an HTML file or a Java Network Launch Protocol (JNLP) file.

In general, a JavaFX program is compiled like any other Java program. However, because of the need for additional support for various execution environments, the easiest way to compile a JavaFX application is to use an Integrated Development Environment (IDE) that fully supports JavaFX programming, such as NetBeans. Just follow the instructions for the IDE you are using.

Alternatively, if you want to compile and test a JavaFX application using the commandline tools, you can easily do so. Just compile and run the application in the normal way, using **javac** and **java**. Be aware that using the command-line compiler neither creates any HTML or JNLP files that would be needed if you want to run the application in a way other than as a stand-alone application, nor does it create a JAR file for the program. To create these files, you need to use a tool such as **javafxpackager**.

The Application Thread

In the preceding discussion, it was mentioned that you cannot use the **init()** method to construct a stage or scene. You also cannot create these items inside the application's constructor. The reason is that a stage or scene must be constructed on the *application thread*. However, the application's constructor and the **init()** method are called on the main thread, also called the *launcher thread*. Thus, they can't be used to construct a stage or scene. Instead, you must use the **start()** method, as the skeleton demonstrates, to create the initial GUI because **start()** is called on the application thread. Furthermore, any changes to the GUI currently displayed must be made from the application thread. Fortunately, in JavaFX, events are sent to your program on the application thread. Therefore, event handlers can be used to interact with the GUI. The **stop()** method is also called on the application thread.

A Simple JavaFX Control: Label

The primary ingredient in most user interfaces is the control because a control enables the user to interact with the application. As you would expect, JavaFX supplies a rich assortment of controls. The simplest control is the label because it just displays a message, which, in this example, is text. Although quite easy to use, the label is a good way to introduce the techniques needed to begin building a scene graph.

The JavaFX label is an instance of the **Label** class, which is packaged in **javafx.scene.control**. **Label** inherits **Labeled** and **Control**, among other classes. The **Labeled** class defines several features that are common to all labeled elements (that is, those that can contain text), and **Control** defines features related to all controls.

Label defines three constructors. The one we will use here is:

`Label(String str)`

Here, *str* is the string that is displayed.

Once you have created a label (or any other control), it must be added to the scene's content, which means adding it to the scene graph. To do this, you will first call **getChildren()** on the root node of the scene graph. It returns a list of the child nodes in the form of an **ObservableList<Node>**. **ObservableList** is packaged in **javafx.collections**, and it inherits **java.util.List**, which means that it supports all of the features available to a list as defined by the Collections Framework. Using the returned list of child nodes, you can add the label to the list by calling **add()**, passing in a reference to the label.

The following program puts the preceding discussion into action by creating a simple JavaFX application that displays a label:

JavaFXLabelDemo.java

```
// Demonstrate a JavaFX label.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate a JavaFX label.");

        // Use a FlowPane for the root node.
```

```
FlowPane rootNode = new FlowPane();

// Create a scene.
Scene myScene = new Scene(rootNode, 300, 200);

// Set the scene on the stage.
myStage.setScene(myScene);

// Create a label.
Label myLabel = new Label("This is a JavaFX label");

// Add the label to the scene graph.
rootNode.getChildren().add(myLabel);

// Show the stage and its scene.
myStage.show();
}
}
```



Figure 2: Output of JavaFXLabelDemo.java

Using Buttons and Events

Although the program in the preceding section presents a simple example of using a JavaFX control and constructing a scene graph, it does not show how to handle events. As you know, most GUI controls generate events that are handled by your program. For example, buttons, check boxes, and lists all generate events when they are used. In many ways, event handling in JavaFX is similar to event handling in Swing or the AWT, but it's

more streamlined. Therefore, if you already are proficient at handling events for these other two GUIs, you will have no trouble using the event handling system provided by JavaFX.

One commonly used control is the button. This makes button events one of the most frequently handled. Therefore, a button is a good way to demonstrate the fundamentals of event handling in JavaFX. For this reason, the fundamentals of event handling and the button are introduced together.

Event Basics

The base class for JavaFX events is the **Event** class, which is packaged in **javafx.event**. **Event** inherits **java.util.EventObject**, which means that JavaFX events share the same basic functionality as other Java events. Several subclasses of **Event** are defined. The one that we will use here is **ActionEvent**. It handles action events generated by a button. In general, JavaFX uses what is, in essence, the delegation event model approach to event handling. To handle an event, you must first register the handler that acts as a listener for the event. When the event occurs, the listener is called. It must then respond to the event and return. In this regard, JavaFX events are managed much like Swing events, for example. Events are handled by implementing the **EventHandler** interface, which is also in **javafx.event**. It is a generic interface with the following form:

```
interface EventHandler<T extends Event>
```

Here, **T** specifies the type of event that the handler will handle. It defines one method, called **handle()**, which receives the event object as a parameter. It is shown here:

```
void handle(T eventObj)
```

Here, *eventObj* is the event that was generated. Typically, event handlers are implemented through anonymous inner classes or lambda expressions, but you can use stand-alone classes for this purpose if it is more appropriate to your application (for example, if one event handler will handle events from more than one source).

Although not required by the examples in this chapter, it is sometimes useful to know the source of an event. This is especially true if you are using one handler to handle events from different sources. You can obtain the source of the event by calling **getSource()**, which is inherited from **java.util.EventObject**. It is shown here:

```
Object getSource( )
```

Other methods in **Event** let you obtain the event type, determine if the event has been consumed, consume an event, fire an event, and obtain the target of the event. When an

event is consumed, it stops the event from being passed to a parent handler. One last point: In JavaFX, events are processed via an *event dispatch chain*. When an event is generated, it is passed to the root node of the chain. The event is then passed down the chain to the target of the event. After the target node processes the event, the event is passed back up the chain, thus allowing parent nodes a chance to process the event, if necessary. This is called *event bubbling*. It is possible for a node in the chain to consume an event, which prevents it from being further processed.

NOTE Although not used in this introduction to JavaFX, an application can also implement an *event filter*, which can be used to manage events. A filter is added to a node by calling `addEventFilter()`, which is defined by `Node`. A filter can consume an event, thus preventing further processing.

Introducing the Button Control

In JavaFX, the push button control is provided by the **Button** class, which is in **javafx.scene.control**. **Button** inherits a fairly long list of base classes that include **ButtonBase**, **Labeled**, **Region**, **Control**, **Parent**, and **Node**. If you examine the API documentation for **Button**, you will see that much of its functionality comes from its base classes. Furthermore, it supports a wide array of options. However, here we will use its default form. Buttons can contain text, graphics, or both. In this chapter, we will use text-based buttons. An example of a graphics-based button is shown in the next chapter.

Button defines three constructors. The one we will use is shown here:

```
Button(String str)
```

In this case, *str* is the message that is displayed in the button.

When a button is pressed, an **ActionEvent** is generated. **ActionEvent** is packaged in **javafx.event**. You can register a listener for this event by using **setOnAction()**, which has this general form:

```
final void setOnAction(EventHandler<ActionEvent> handler)
```

Here, *handler* is the handler being registered. As mentioned, often you will use an anonymous inner class or lambda expression for the handler. The **setOnAction()** method sets the property **onAction**, which stores a reference to the handler. As with all other Java event handling, your handler must respond to the event as fast as possible and then return. If your handler consumes too much time, it will noticeably slow down the application. For lengthy operations, you must use a separate thread of execution.

Demonstrating Event Handling and the Button

The following program demonstrates event handling. It uses two buttons and a label. Each time a button is pressed, the label is set to display which button was pressed.

JavaFXEventDemo.java

```
// Demonstrate JavaFX events and buttons.
import javafx.application.*;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

    Label response;

    public static void main(String[] args) {
        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate JavaFX Buttons and Events.");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
```

```
Scene myScene = new Scene(rootNode, 300, 100);

// Set the scene on the stage.
myStage.setScene(myScene);

// Create a label.
response = new Label("Push a Button");

// Create two push buttons.
Button btnAlpha = new Button("Alpha");
Button btnBeta = new Button("Beta");

// Handle the action events for the Alpha button.
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Alpha was pressed.");
    }
});

// Handle the action events for the Beta button.
btnBeta.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Beta was pressed.");
    }
});

// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);

// Show the stage and its scene.
myStage.show();
}
}
```

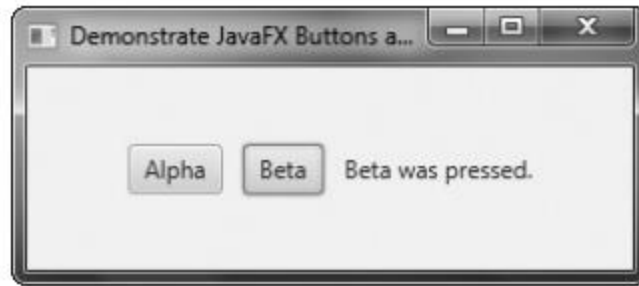


Figure 3: Output of JavaFXEventDemo.java