11 Best of both worlds: using JavaFX from Java

This chapter covers

- Mixing JavaFX into Java programs
- Calling JavaFX Script as a scripting language
- Defining our Java app's Swing UI in JavaFX
- Adding JavaFX to the Java classpath

In previous chapters we saw plenty of examples demonstrating Java classes being used from JavaFX Script. Now it's time for an about-face; how do we call JavaFX Script from Java?

Because JavaFX Script compiles directly to JRE-compatible bytecode, it might be tempting to assume we can treat JavaFX-created classes in much the same way we might treat compiled Java classes or JARs. But this would be unwise. There are enough differences between the two languages for assumption making to be a dangerous business. For example, JavaFX Script uses a declarative syntax to create new objects; any constructors in the bytecode files are therefore a consequence of compiler implementation, not the JavaFX Script code. We can't guarantee future JavaFX Script compilers will be implemented the same way; constructors present in classes

written by one JFX compiler might vanish or change in the next. So in this chapter we'll look at how to get the two languages interacting in a manner that doesn't depend on internal mechanics.

A reasonable question to ask before we get going is, "When is an application considered Java, and when is it JavaFX Script?" Suppose I write a small JavaFX Script program that relies on a huge JAR library written in Java; is my program a Java program that uses JavaFX Script or a JavaFX Script program that relies on Java? Which is the *primary* language?

Obviously, there are different ways of measuring this, but for the purposes of this chapter the primary language is the one that forms the entry point into the application. In our scenario it's JavaFX Script that runs first, placing our hypothetical application unambiguously into the category of a JavaFX application that uses Java. We've seen plenty of examples of this type of program already. In the pages to come we'll focus exclusively on the flip side of the coin: bringing JavaFX Script into an already-running Java program.

11.1 Different styles of linking the two languages

There are two core ways we might employ JavaFX Script in our Java programs.

- *As a direct alternative to Java*—We might enjoy JavaFX Script's declarative syntax so much we decide to write significant portions of our Java program's UI in it. In this scenario JavaFX Script plays no part in the final binary release (although JavaFX APIs may be used). Everything is compiled to bytecode, and JFX is merely being used as a tool to write part of the source code.
- *As a runtime scripting language*—We might want to add scripting capabilities to our Java application and decide that JavaFX Script is a suitable language to achieve this. In this scenario JavaFX Script is an active part of the final product, providing a scriptable interface to control the application as it runs.

The project we'll develop in this chapter demonstrates both uses.

11.2 Adventures in JavaFX Script

What we need now is an interesting project that demands both types of script usage, something like a simple adventure game. We can use JavaFX Script as a Java replacement to create some of the UI and as a scripting language for the game events.

The technique of breaking up an application into a *core* engine and a series of *lightweight* event scripts is well known in the video games industry, but it's also becoming increasingly common in productivity applications too. It allows programs like word processors and graphics tools to open themselves up to enhancement and customization without releasing their source code. Mostly it's the larger, more sophisticated applications that benefit from script-ability like this; thus, for the purposes of this book, it's probably best we stick to a simple game—although the techniques are exactly the same, no matter what the scale or purpose of the program.

Creating the graphics for an adventure game can take longer than writing the actual game code itself. Lucky, then, that your humble author toyed with an isometric game engine way back in the days of 16-bit consoles. Even luckier, a ready-made palette of isometric game tiles (painstakingly crafted in Deluxe Paint, I recall) survived on an old hard drive (see figure 11.1) ready to be plundered. The graphics are a little retro in appearance but will serve our needs well. Remember, kids, it's good to recycle!

The engine we'll use will be constructed in Java and just about functional enough to plug in the JavaFX Script code we want to use with it. Since this is not a Java book, I won't be reproducing the Java source code in full within these pages. Indeed, we won't even be looking at how the engine works; the game engine is a means to an end—a sample application we can use as a test bed for our JavaFX Script integration. Instead, we'll focus on the fragments binding JavaFX Script into the Java.

Download the source

The majority of the project's Java code is not reproduced in this chapter (this is a JavaFX book, after all!). You can download the full source and the associated graphics files from the book's website. The source is fully annotated; each section relating to working with JavaFX Script is clearly labeled. Try searching for the keyword *JavaFX*.

http://manning.com/JavaFXinAction/

For the sake of flexibility many video games are developed in two distinct parts. A programmer will build a *game engine*, which drives the game based on the graphics, sounds, and level/map data the designers feed into it. This data is often created with specially written tools for that particular game engine. It's a process not unlike the programmer/designer workflow we touched on with the Enigma applet project.

Figure 11.1 A simple Java adventure game engine, using an isometric view. The control panel at the foot of the window, as well as the in-game events, will be written using JavaFX Script.

Our simple Java isometric game engine employs a grid-based map. The game environment is broken up into rooms, and each room is a grid of stacked images, carefully placed to give an isometric 3D effect. Take a look at figure 11.2 for a visual representation of how they are arranged.

Unlike the maze we developed previously, the game map isn't hardcoded into the source. A simple data file is used, which the game engine reads to build the map. Obviously, it's not enough for the game's designer to build a static map; she needs to add often quite complex interactions to model the game events. This is when the programmer reaches for a scripting engine.

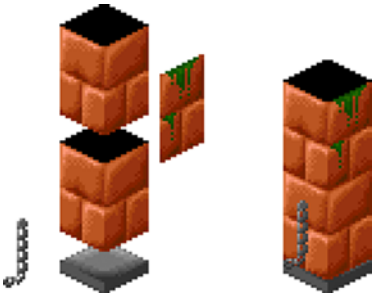

Figure 11.2 Each cell in the game environment is created from up to five images: one floor tile, two wall tiles, and two faces that modify one side of a wall tile.

11.2.1 Game engine events

Building lightweight scripting into a game engine allows the game designer the freedom she needs to program the game's *story*, without hardcoding details into the game application itself. Our game engine uses JavaFX Script as its scripting engine to support three types of game events:

- An event can run whenever the player leaves a given cell.
- An event can run whenever the player enters a given cell.
- We can define *actions*, which the player triggers by standing on a given cell and pressing the spacebar on the keyboard, for example, when the player wants to throw a wall-mounted switch.

Our game engine uses a large text file, which describes each room, including the width and height of its area (in cells), the tile images that appear in each cell, and the door locations connecting rooms. More important, it also includes the event code, written in JavaFX Script. We can see how a part of it might look in listing 11.1.

Listing 11.1 A fragment of the game data file

```
#ROOM 1 6 6
#TITLE The 2nd Room
0000010f06 000C010f06 0E00010f06 0008010f06 004c010f06 0008010f06
0000010f06 0000000006 0000002b06 0000000006 0000000006 0045000006
0000010f06 0000000006 0000002a06 0000000006 0000000003 0000000008
0000071106 0000000006 0000000005 0000000006 0000000003 0000000003
0000081006 0000001e06 0000000003 0000000003 0000000003 0000000003
000A010f06 0000000006 0000000007 0000002c06 0000002d06 0000000004
#LINK 1 5 1 1 to 0 1 2 1
#REPAINT 000000 001000 001000 000000 010000 000110
#END

#SCRIPT 1 4 1 action
def st = state as jfxia.chapter11.State;
```

Grid of tile graphics

Door link

Action event as JavaFX Script

```
if(st.getPlayerFacing()==0)
    st.setRoomCell(1,4,0 , -1,-1,-1,0x4e,-1);
    st.setRoomCell(0,6,1 , -1,0,-1,-1,-1);
}
#END
```

We have only a fragment of the data file shown here, detailing just one room. The syntax is one I made up myself to quickly test the engine; if this were a serious project, I might have used XML instead. The whole file is parsed by the game engine upon startup. We're not interested in the details, but for the record, the `#ROOM` line declares this data to be for room 1 (rooms start at 0) and 6 x 6 cells in size, the `#TITLE` line gives the room a name, and a `#LINK` line connects room 1 cell (5,1) east, with room 0 cell (1,2) east. The `#REPAINT` line is a series of flags that help optimize screen updates. Figure 11.3 shows the room in question.

The important part is the `#SCRIPT` block, attached to room 1 cell (4,1) as an *action* event type, meaning it will run when the player presses the spacebar. The script checks to see whether the player is facing north (0 means north), and if so it changes a tile at cell (4,0), flipping the switch mounted on the wall into the up position. It also changes a tile in another room, removing an obstruction.

The script does all of these things by calling functions on an object named `st`, which is a reference (cast as type `jfxia.chapter11.State`) to another object called `state`. The obvious question is, "Where does `state` come from?" I'm sure it won't come as any surprise to learn it's an object dropped into our JavaFX Script environment by the Java code that calls our script. To investigate further we need to check out that Java code, which is precisely where we'll head next.

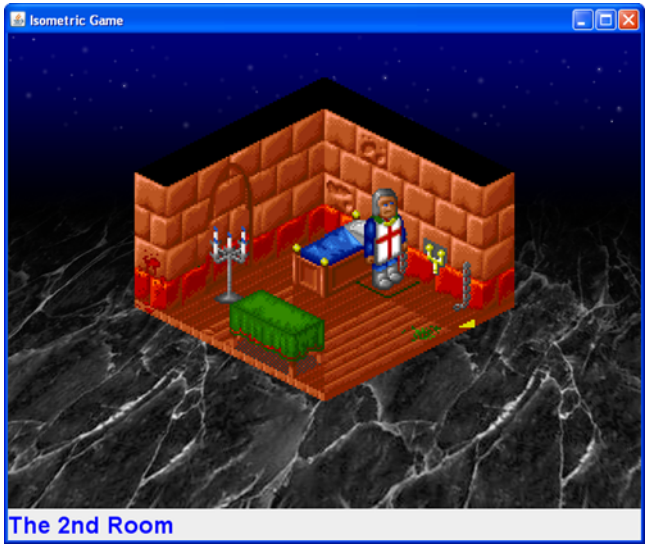

Figure 11.3 This is room 1 (room IDs start at 0), which the fragment of data file in listing 11.1 refers to. The player stands on cell (3,1), in front of him is the event cell (4,1), and beyond that the door link cell (5,1).

11.2.2 Calling the JavaFX Script event code from Java

Java SE version 6 supports JSR 223. For those who don't recognize the name, JSR 223 is the Java Specification Request for running scripting languages within Java programs. By implementing something called a *scripting engine* (not to be confused with our game engine) a language can make itself available to running Java programs. Not only can Java run scripts written in that language, but it may also be able to share objects with its scripts.

JSR 223 uses what's known as the *service provider mechanism*. To use the scripting engine (and the language it implements) all we need do is include the engine's JAR on Java's classpath, and the engine will be found when requested. Even though we generally work with JavaFX Script as a compiled language, JavaFX comes with a scripting engine. It lives in the javafxc.jar file, inside the lib directory of your JavaFX SDK installation.

Important: Getting the classpath right

When you run a JavaFX program using the `javafx` command (directly or via an IDE), the standard JavaFX JAR files are included on the *classpath* for you. When you invoke JavaFX code from Java, however, this won't happen. You need to manually add the necessary JARs to the classpath. If you explore your JavaFX installation, you'll see subdirectories within lib for each profile. The files you need depend on which features of JavaFX you use; to get our game running I found I needed to include the following from the JavaFX 1.2 SDK:

shared/javafxc.jar (for the JavaFX Script JSR 223 scripting engine)
shared/javafxrt.jar
desktop/decora-runtime.jar
desktop/javafx-anim.jar
desktop/javafx-geom.jar
desktop/javafx-sg-common.jar
desktop/javafx-sg-swing.jar
desktop/javafx-ui-common.jar
desktop/javafx-ui-desktop.jar
desktop/javafx-ui-swing.jar

Tip: if figuring out which JARs you need is too painful for you, why not set the `java.ext.dirs` property when you run your JRE, pointing it at the necessary library directories within JavaFX? (Of course, while this solution is fine on our developer box, it doesn't easily deploy to regular users' computers, without wrapping our program in a startup script.)

```
java -Djava.ext.dirs=<search path> MyJavaClass
```

Once the javafxc.jar file is on the classpath, we need to know how to invoke its scripting engine, and that's precisely what listing 11.2 does. It shows fragments of the `Map` class, part of the Java game engine I implemented for this project. This listing shows how to set up the scripting engine; we'll deal with actually running scripts later.

Listing 11.2 Map.java: calling JavaFX Script from Java

```
import javax.script.ScriptEngineManager;
import com.sun.javafx.api.JavaFXScriptEngine;

// ...

boolean callEnterEventScript(int kx,int ky) {
    return callEventScript(currentRoom.enter , kx,ky);
}
boolean callExitEventScript(int kx,int ky) {
    return callEventScript(currentRoom.exit , kx,ky);
}
boolean callActionEventScript(int kx,int ky) {
    return callEventScript(currentRoom.action , kx,ky);
}

private boolean callEventScript(HashMap<Integer,String>hash,
int kx,int ky) {
    int x=state.playerX , y=state.playerY , f=state.playerFacing;
    Room r=currentRoom;

    int key = kx*100+ky;
    String script;
    if(hash.containsKey(key)) {
        script = hash.get(key);
        try {
            ScriptEngineManager manager =
                new ScriptEngineManager();
            JavaFXScriptEngine jfxScriptEngine =
                (JavaFXScriptEngine)manager
                .getEngineByName("javafx");

            jfxScriptEngine.put("state",state);

            jfxScriptEngine.eval(script);
        }catch(Exception e) {
            e.printStackTrace();
            System.err.println(script);
        }
    }

    return !(state.playerX==x && state.playerY==y &&
        state.playerFacing==f && currentRoom==r);
}
```

Import JavaFX scripting engine

Three different event types

Handle each game event

Event code from game data

Service provider finds JFX engine

Make state available

Run script

Remember, this is not the complete source file—just the bits relating to JSR 223.

At the head of the listing we import the classes required for Java to talk to the JavaFX Script scripting engine. The first import, `javax.script.ScriptEngineManager`, is the class we'll use to *discover* (get a reference to) the JavaFX Script engine. The second

import, `com.sun.javafx.api.JavaFXScriptEngine`, is the JavaFX Script scripting engine itself.

The bulk of the code is taken from the body of the `Map` class. Each of the three event types supported has its own method. The game engine uses three hash tables to store the events. Each method defers to a central event handling method, `callEventScript()`, passing the necessary hash table for the current room, along with the cell x/y position relating to the event (typically the player's location).

The `callEventScript()` method combines the x and y positions into a single value, which it uses as a key to extract the JavaFX Script code attached to that cell. The script code is loaded into a `String` variable called, appropriately enough, `script`. This variable now holds raw JavaFX Script source code, like we saw in listing 11.1, earlier. To run this code we use a scripting engine, and that's what we see in the middle of the `callEventScript()` method.

```
ScriptEngineManager manager = new ScriptEngineManager();
JavaFXScriptEngine jfxScriptEngine =
    (JavaFXScriptEngine)manager.getEngineByName("javafx");
```

First we use Java's `ScriptEngineManager` to get a reference to JavaFX's `JavaFX-ScriptEngine`. You can see that we first create a new manager and then ask it to find (using the service provider mechanism) a scripting engine that matches the token `javafx`. Assuming the necessary JavaFX Script JAR file is on the classpath, the manager should get a positive match.

```
jfxScriptEngine.put("state",state);
```

Having acquired a reference to a JavaFX Script scripting engine, we add a Java object, `state`, into the engine's runtime environment. We encountered this variable in the event code of listing 11.1. The `state` object is where our game engine holds status data like the position of the player and references to map data. It also provides methods to modify this data, which is why it's being shared with the JavaFX Script environment. We can share as many Java objects as we please like this and choose what names they appear under in the script environment (in our code, however, we stick with `state`).

```
jfxScriptEngine.eval(script);
```

The call to `eval()`, passing the JavaFX Script code, causes the code to run. An exception handler is needed around the `eval()` call to catch any runtime errors thrown by the script. If all goes according to plan, though, we will have successfully called JavaFX Script from within a Java program, *without* relying on any compiler implementation detail to link the two.

So that's one way to hook JavaFX Script into Java, but what about the other way? This is where we'll head next.

Problems with JavaFX 1.2?

When I wrote this project against JavaFX 1.1, I created a single `JavaFXScriptEngine` object in the constructor, passing in the reference to `state`, and reused it for each `eval()` call. Some deep changes were made to the JSR 223 implementation for JavaFX 1.2, and I found this no longer worked; `eval()` would always rerun the first script the engine was passed, ignoring the new code passed in. The solution was to create a fresh engine for each script call. I'm not sure if this is a bug in my understanding of JavaFX's JSR 223 code, a bug in the JSR 223 implementation itself, or just a feature! Per Bothner has written a blog entry about the 1.2 changes, so if you're interested, go take a look:

http://per.bothner.com/blog/2009/JavaFX-scripting-changes/

11.3 Adding FX to Java

So far we've looked at how to treat JavaFX Script as a runtime scripting language. As you'll recall, there's another way it can be used in a Java application; we can mix compiled Java and compiled JavaFX Script in the same project, developing different parts of our program in different languages.

The preview release of JavaFX (pre 1.0) featured a `Canvas` class, acting as a bridge between the JavaFX scene graph and AWT/Swing. Sadly, this class was removed from the full release, making it impossible to pass anything except Swing wrapper nodes back to a Java UI. But even without the scene graph, this technique is still useful, for the following reasons:

- It's entirely possible the JavaFX team may add a bridge between the scene graph and AWT/Swing back in at a later release.
- Depending on what you're writing, JavaFX Script's declarative syntax may be useful for building data structures other than just graphics. (Be careful: JavaFX was designed for GUI coding, and as such it likes to run its code on the UI event dispatch thread.)

Using a JavaFX scene graph inside Java: the hack

When JavaFX 1.0 was released, Sun engineer Josh Marinacci blogged about a quick-'n'-dirty hack from fellow Java/JFX team members Richard Bair and Jasper Potts, building a bridge between the JavaFX scene graph and Swing. Unfortunately, this code stopped working when 1.2 arrived, but the open source JFXtras project (created by Stephen Chin) took up the challenge. Their 1.2-compatible solution, however, relies on internal scene graph implementation detail and isn't guaranteed to work in future releases.

http://blogs.sun.com/javafx/entry/how_to_use_javafx_in
http://code.google.com/p/jfxtras/

As an example our project will use a very basic control panel below the main game view. The simple panel is a JavaFX Script–created `SwingLabel`. To see how it looks, see figure 11.4.

Figure 11.4 The panel at the foot of the game's window is written entirely in compiled JavaFX Script.

In the sections that follow we'll look at how to implement this panel in a way that makes it easy for the two languages to interact safely.

11.3.1 The problem with mixing languages

Think about what we're trying to achieve here. We have three languages in play: two high-level languages (Java and JavaFX Script) are both compiled to a common low-level language (bytecode). When we used JSR 223, the two languages were separated through the high-level abstraction of a scripting engine. Effectively the Java program was self-contained, and JavaFX Script code was treated almost as *data* (just like the map data). But in this section we're attempting to use the two languages together, creating two halves of a single program.

JavaFX Script guarantees a high degree of interoperability with Java as part of its language syntax; JavaFX programmers don't have to worry about the internals of class files created from Java code. The same is not true in the opposite direction. Although it seems likely that JavaFX Script functions will translate directly into bytecode methods, this is just an assumption. It may not be true of future (or rival) JavaFX Script compilers. How do we link the two languages in *both* directions, without making any assumptions?

Fortunately, an elegant solution emerges after a little bit of lateral thinking. If JavaFX Script's compatibility with Java classes is guaranteed, but Java's compatibility with JavaFX Script classes is undefined, can we use the former to fix the latter? Yes, we can, using Java interfaces!

11.3.2 The problem solved: an elegant solution to link the languages

If we encode the interactions between our two languages into a Java interface and get the JavaFX Script code to implement (or *extend*, to use the JFX parlance) this interface, we have a guaranteed Java-compatible bridge between the two languages that Java can exploit to communicate with the JavaFX Script software.

Listing 11.3 shows the Java interface created for our game engine. It has only two methods: the first is used to fetch the control panel user interface as a Java

Skinning cats

We can also form a bridge by subclassing a Java class. However, interfaces, with their lack of inherited behavior, generally provide a cleaner (sharper, less complicated) coupling. But, as the saying goes, "there's more than one way to skin a cat." Choose the way that makes sense to you and your current project.

Swing–compatible object (a `JComponent`), and the second is used to interact with the user interface.

Listing 11.3 ControlPanel.java

```
package jfxia.chapter11;

import javax.swing.JComponent;

public interface ControlPanel {
    public JComponent getUI();
    public void setTitle(String s);
}
```

Fetch Java-compatible UI

Interact with UI

Now that you're familiar with the interface and its methods, let's see them in action. Listing 11.4 shows fragments of the `Game` class, written in Java, displaying the first part of how to hook a JavaFX-created user interface control into Java Swing.

Listing 11.4 Game.java (part 1): adding JavaFX UIs to Java code

```
import javax.script.ScriptEngineManager;
import com.sun.javafx.api.JavaFXScriptEngine;

// ...

private ControlPanel ctrlPan;

// ...

ctrlPan = getJavaFX();
ctrlPan.setTitle(state.getCurrentRoomTitle());

JPanel pan = new JPanel(new BorderLayout());
pan.add(mapView,BorderLayout.CENTER);
pan.add(ctrlPan.getUI(),BorderLayout.SOUTH);

//Part 2 is listing 11.5
```

Imports required

Control panel reference

Create and set up

Hook into Java Swing

First we create a class variable to hold our JavaFX object reference, using the `ControlPanel` interface we defined as a bridge. Then comes the meaty part: to plug the two UIs together we use the method `getJavaFX()`, which fetches the `ControlPanel` (details in listing part 2, so be patient!). We call a method on it to set the initial room name; then we use the `ControlPanel.getUI()` interface method to pull a Swing-compatible `JComponent` from the JavaFX Script code and add it into the

southern position of a `BorderLayout` panel. (The other component, added to the center position, is the main game view, in case you're wondering.)

Exactly how the JavaFX object is turned into a Java object is hidden behind the mysterious `getJavaFX()` method, which will surrender its secrets next.

11.3.3 Fetching the JavaFX Script object from within Java

The code that creates the `ControlPanel` class will look very familiar. It's just a variation on the JSR 223 scripting engine code you saw earlier in listing 11.2.

Listing 11.5 is the second half of our `Game` class fragments. It shows the method `getJavaFX()` using the JavaFX Script scripting engine.

Listing 11.5 Game.java (part 2): adding JavaFX user interfaces to Java code

```
private ControlPanel getJavaFX() {
    ScriptEngineManager manager = new ScriptEngineManager();
    JavaFXScriptEngine jfxScriptEngine =
        (JavaFXScriptEngine)manager.getEngineByName ("javafx");
    try {
        return (ControlPanel)jfxScriptEngine.eval (
            "import jfxia.chapter11.jfx.ControlPanelImpl;\n"+
            "return ControlPanelImpl{};"
        );
    }
    catch(Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Since we're going to use the scripting engine only once, we lump all the code to initialize the engine and call the script into one place. The script simply returns a declaratively created JavaFX Script object of type `ControlPanelImpl`, which (you can't tell from this listing, but won't be shocked to learn) extends our `ControlPanel` interface. The object returned by the script provides the return value for the `getJavaFX()` method.

Figure 11.5 demonstrates the full relationship. Once the `ControlPanelImpl` object is created, Java and JavaFX Script communicate only via a common interface. JavaFX Script's respect for Java interfaces means this is a safe way to link the two. However, JavaFX Script's lack of support for constructors means the `ControlPanelImpl` object must still be created using JSR 223.

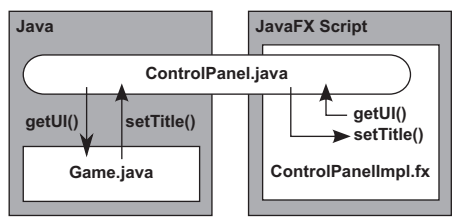

Figure 11.5 Java's `Game` class and the JavaFX Script `ControlPanelImpl.fx` class communicate via a Java interface, `ControlPanel.java`.

Let's look at `ControlPanelImpl`. Because it's written in JavaFX Script, the code is presented in its entirety (for those suffering JavaFX withdrawal symptoms) in listing 11.6.

Listing 11.6 ControlPanelImpl.fx

```
package jfxia.chapter11.jfx;

import javafx.ext.swing.SwingComponent;
import javafx.ext.swing.SwingHorizontalAlignment;
import javafx.ext.swing.SwingLabel;
import javafx.ext.swing.SwingVerticalAlignment;
import javafx.scene.CustomNode;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;

import javax.swing.JComponent;
import jfxia.chapter11.ControlPanel;

public class ControlPanelImpl extends CustomNode, ControlPanel {
    public-init var text:String;
    var myNode:SwingComponent;
    var myColor:Color = Color.BLACK;

    public override function create() : Node {
        myNode = SwingLabel {
            text: bind text;
            font:
                Font.font("Helvetica",FontWeight.BOLD,24);
            horizontalTextPosition:
                SwingHorizontalAlignment.CENTER;
            verticalTextPosition:
                SwingVerticalAlignment.CENTER;
            foreground: Color.BLUE;
        }
    }

    public override function getUI() : JComponent {
        myNode.getJComponent();
    }

    public override function setTitle(s:String):Void {
        text = s;
    }
}
```

Required for Java coupling

Subclass our Java interface

Create and store node

ControlPanel interface methods/functions

The script looks just like a standard Swing node. At the foot of the listing we see two functions that implement the `ControlPanel` interface. The first, `getUI()`, pulls the Swing `JComponent` from our node using the `getJComponent()` function supported by the Swing wrappers in the `javafx.ext.swing` package. The second function, `setTitle()`, allows Java to change the displayed text string of the control panel.

And that's it! We now have a successful morsel of JavaFX code running happily inside a Java Swing application. Not hard, when you know how!

Important: Compiling the code

This is a minor point but an important one: the JavaFX Script class relies on the Java interface, so whatever process or IDE we use to build the source code, we need to ensure the Java code is compiled first. The Java code doesn't need the interface implementation to be available when it builds (the whole idea of interfaces is so the implementation can be plugged in later). The JavaFX Script code needs the interface class to be available, and on the classpath, for it to build.

Using an interface has one extra advantage: providing we haven't changed the interface itself, we don't need to compile both sides of the application every time the code changes. If all our modifications are restricted to the Java side, we have to recompile only the Java, leaving the JavaFX Script classes alone. Likewise, the reverse is true if all the changes are restricted to the JavaFX Script side.

You could make use of this natural separation when designing your own projects, perhaps assigning the Java and the JavaFX Script code to different teams of developers, coding independently against interfaces that bridge their work.

11.4 Summary

In this chapter we've covered quite a simple skill, yet a sometimes useful one. Bringing JavaFX Script code into an existing Java program is something that needs to be done carefully and without assumptions about how a given JavaFX compiler works. Fortunately Java 6's scripting engine support, coupled with JavaFX Script's respect for the Java language's class hierarchy mechanism, makes it possible to link the two languages in a clean way. It may not be something we need to do every day, but it's nice to know the option is there when we need it.

Using scripting languages from within Java certainly gives us great power. (Don't forget, "With great power there must also come great responsibility!" as Stan Lee once wrote.) It allows us to develop different parts of our application in languages more suitable to the task at hand. Sooner rather than later, I hope, the JavaFX team will add a working bridge between the scene graph and AWT/Swing, so we can exploit JavaFX to the fullest within existing Java applications. JavaFX would surely give us a powerful tool in slashing development times of Java desktop apps and reducing the frequency of UI bugs.

This chapter brings the book to a close. We began in chapter 1 with excited talk about the promise of rich internet applications and the power of domain-specific languages, worked our way through the JavaFX Script language, learned how to use binds, manipulated the scene graph, took a trip to the movies, looked through our photo collection, sent some secret codes from within a web browser, got lost in a mobile maze, and ended up learning how to have the best of both worlds. What a journey! I hope it's been fun.

At the start of the book I stated that my mission was not to reproduce the API documentation blow for blow. The book was written against JavaFX SDK 1.2, and JavaFX still has plenty of room for growth in future revisions; my goal was to give you a good grounding in the concepts and ideas that surround JavaFX, so you could take future enhancements in your stride. The final chapters, as I'm sure you noticed, became more speculative, revealing what might be coming up in future revisions of the JFX platform. Any speculation was based on public material from Sun and the JavaFX team, so I hope there won't be too many surprises after the book goes to press.

Although practicality meant none of the projects in this book were as mind blowing as they could have been, I hope they gave you plenty to think about. Download the source code, play with it, add to it, and fill in the bits I didn't have the space to do. Then try building your own apps. Increasingly, modern applications are featuring sumptuous graphical effects and rich multimedia, and JavaFX gives you the power to compete with the best of them. Push your skills to the max, amaze yourself, amaze your friends, amaze the world!

Above all, have fun!